



# Contourlijnextractie en weergave in interactieve 3D-toepassingen

Jeroen Baert

Thesis voorge dragen tot het behalen van de graad van Master in de ingenieurswetenschappen: computerwetenschappen, optie Mens-Machine Communicatie

**Promotor:**

Prof. Dr. Ir. Ph. Dutré

**Assessoren:**

Prof. Dr. E. Duval  
Dr. Ir. K. Driessens

**Begeleiders:**

Dr. B. J. Brown  
Ir. R. Schoukens

Academiejaar 2009 – 2010

© Copyright K.U.Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor(en) als de auteur(s) is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail [info@cs.kuleuven.be](mailto:info@cs.kuleuven.be).

Voorafgaande schriftelijke toestemming van de promotor(en) is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

# Voorwoord

Een thesis schrijf je nooit alleen: ik wens verschillende mensen persoonlijk te bedanken voor hun hulp bij het tot stand brengen van deze masterproef. Bovendien beschouw ik een thesis als sluitstuk van een onvergetelijke periode uit mijn leven, en wens ik daar eveneens enkele mensen voor te bedanken.

Vooraleerst bedank ik mijn promotor Prof. Dr. Ir. Ph. Dutré, niet alleen voor het vertrouwen en de begeleiding, maar eveneens voor het aanwakkeren van mijn interesse voor computer graphics in de verschillende vakken die ik bij hem mocht volgen. Ik bedank eveneens mijn begeleiders Dr. B. J. Brown en Ir. R. Schoukens voor hun nuttige suggesties en professionele sturing, zowel theoretisch als technisch, gedurende het hele academiejaar.

Ik bedank mijn ouders voor alle kansen die ze me gegeven hebben aan deze universiteit, alsook mijn broer Pieter en zus Elien voor de ondersteuning door de jaren heen. Een familie hebben waarop men kan terugvallen is onbetaalbaar.

Mijn vriendin Kelly doorstond vele nachten vol toetsenbord-geratel: ik wil haar bedanken voor de liefde en vriendschap tijdens de mooie momenten, alsook voor de onvoorwaardelijke steun tijdens de moeilijke. Zonder haar had ik het niet gekund.

Voor technische ondersteuning en tips kon ik steeds terecht bij VTK-Revue. In het bijzonder wens ik Ruben, Tim en Pieter te bedanken. Ook bedank ik alle leden van de groep voor de onvergetelijke voorstellingen: ervaringen voor het leven.

De nodige ontspanning en fysieke/mentale uitlaatklep vond ik gedurende de jaren bij studentenimprovisatieteam Preparee: ik beschouw het als een eer om deel uitgemaakt te hebben van deze unieke groep creatievelingen, en wens hen als trotse ex-kapitein alleen het beste toe. In het bijzonder wens ik Catheline, Dieter en Pieterjan te bedanken.

Men zegt vaak dat een mens is de som van zijn vrienden: zonder te claimen dat de som hiermee volledig is wens ik van hen eveneens Vincent, Wanda en Benjamin te bedanken.

Indien ik in dit dankwoord iemand zou vergeten zijn is een *mea culpa* alvast op zijn plaats.

Leuven, Juni 2010

Jeroen Baert

# Inhoudsopgave

<b>Voorwoord</b>	<b>i</b>
<b>Inhoudsopgave</b>	<b>ii</b>
<b>Samenvatting</b>	<b>iv</b>
<b>Lijst van figuren</b>	<b>v</b>
<b>Lijst van tabellen</b>	<b>vii</b>
<b>Lijst van codeblokken</b>	<b>viii</b>
<b>Lijst van afkortingen en symbolen</b>	<b>ix</b>
<b>1 Inleiding</b>	<b>1</b>
1.1 Lijntekeningen . . . . .	1
1.2 Computer Graphics . . . . .	1
1.3 Fotorealistisch versus Niet-Fotorealistisch renderen . . . . .	3
1.4 Doel . . . . .	4
<b>2 Contourlijnen: Situering en Theorie</b>	<b>5</b>
2.1 Contouren en Suggestieve Contouren: intuïtief . . . . .	5
2.2 Contouren: definitie . . . . .	8
2.3 Suggestieve contouren: definitie . . . . .	9
2.4 Besluit . . . . .	15
<b>3 Methodes voor Contourlijnextractie</b>	<b>16</b>
3.1 Object Space en Image Space algoritmes . . . . .	16
3.2 Eigenschappen . . . . .	17
3.3 Hybride algoritmes . . . . .	20
3.4 Besluit . . . . .	21
<b>4 Object Space Algoritmes: CPU</b>	<b>22</b>
4.1 Technologie en begrippen . . . . .	22
4.2 Contouren . . . . .	25
4.3 Suggestieve contouren . . . . .	30
4.4 Performantieverbeteringen . . . . .	38
4.5 Stochastische technieken . . . . .	42
4.6 Temporele coherentie verbeteren . . . . .	48
4.7 Aanvulling: Suggestieve highlights . . . . .	53

<b>5 Object Space Algoritmes: GPU</b>	<b>59</b>
5.1 Technologie en begrippen . . . . .	59
5.2 Taking it to the GPU: Voordelen / Nadelen . . . . .	61
5.3 Suggestieve contouren via <i>Texture Mapping</i> . . . . .	62
5.4 Suggestieve contouren via <i>Shaders</i> . . . . .	68
<b>6 Image Space Algoritmes</b>	<b>78</b>
6.1 Technologie en begrippen . . . . .	78
6.2 Contouren via Threshold filter . . . . .	80
6.3 Contouren via Sobel filter . . . . .	85
6.4 Suggestieve contouren via radiale filter . . . . .	91
<b>7 Resultaten en Besluit</b>	<b>98</b>
7.1 Eigen inbreng . . . . .	98
7.2 CPU-gebaseerde algoritmes . . . . .	99
7.3 GPU-gebaseerde algoritmes . . . . .	102
<b>A Poster</b>	<b>107</b>
<b>B IEEE-stijl artikel</b>	<b>109</b>
<b>Bibliografie</b>	<b>116</b>

# Samenvatting

In lijntekeningen maken artiesten/ontwerpers gebruik van verschillende soorten contourlijnen om uiteenlopende eigenschappen van het afgebeelde object weer te geven. We willen deze contourlijnen mathematisch definiëren, berekenen en weergeven met computer graphics-technieken. Het doel is om aan interactieve snelheden complexe modellen te kunnen renderen door enkel contourlijnen te gebruiken: dit is een vorm van niet-fotorealistisch renderen, kortweg NPR.

In dit werk worden verschillende bestaande methodes geïmplementeerd en kritisch vergeleken. Op basis van deze analyse worden vervolgens verbeteringen en nieuwe technieken voorgesteld, die gebruik maken van GPU-specifieke functionaliteit om interactieve snelheden te kunnen garanderen. Dit werk introduceert onder meer een nieuw Object Space, shader-gebaseerd algoritme om contourlijnen en suggestieve contourlijnen aan hoge snelheid te renderen.

In conclusie wordt gesteld dat zowel op het vlak van CPU-gebaseerde als op het vlak van GPU-gebaseerde algoritmes er nog ruimte voor verbetering is, zij het elk op een specifieke manier die rekening houdt met de beperkingen van de verschillende soorten algoritmes.

**Keywords:** Contours, Suggestive Contours, NPR, Interactive, 3D, Rendering

# Lijst van figuren

2.1 Lijntekening <i>Kunsthalle</i> , A. Dürer (1505) . . . . .	6
2.2 Render van het Olifant-model met contouren en suggestieve contouren . . . . .	6
2.3 Render van Golfbal-model met contouren en suggestieve contouren . . . . .	8
2.4 De Contour Generator . . . . .	9
2.5 De curvatuur van een curve $\mathbf{C}$ in een punt $\mathbf{p}$ wordt bepaald als de inverse van de straal van de best aansluitende cirkel. . . . .	10
2.6 De richtingsvector $\mathbf{w}$ en de radiale curvatuur . . . . .	10
2.7 Voorbeeld van suggestieve contourlijnpunten . . . . .	11
2.8 Suggestieve Contour Generator . . . . .	12
2.9 Derde definitie van de Suggestieve Contour Generator: illustratie . . . . .	14
2.10 Het belang van de $D_{\mathbf{w}}\kappa_r$ test. . . . .	14
3.1 Object Space / Image Space algoritmes: schema . . . . .	17
3.2 Lijndikte probleem bij Image Space algoritmes . . . . .	18
3.3 View-dependent Level of Detail bij Image Space algoritmes . . . . .	20
4.1 Grafiek met performantievergelijking van algoritmes 4.1 en 4.2. . . . .	27
4.2 Vergelijking resultaten algoritme 4.1 en 4.2 . . . . .	28
4.3 Een probleem in algoritme 4.2: Lussen in de contourlijn. . . . .	29
4.4 Problemen met temporele coherentie in algoritme 4.2 . . . . .	29
4.5 Verschillende manieren om contourlijnen te vormen in algoritme 4.3 . .	35
4.6 Rendering van het Brains-model met contouren en suggestieve contouren	36
4.7 Microcontouren bij toepassing van algoritme 4.3 . . . . .	37
4.8 Gebieden met Positieve Gaussiaanse curvatuur $K$ . . . . .	39
4.9 Onzichtbare suggestieve contouren bij een torus. . . . .	40
4.10 Performantieverbeteringen bij algoritme 4.2 en algoritme 4.3 . . . . .	42
4.11 Benchmarkresultaten bij het toepassen van basisalgoritme 4.4 . . . . .	44
4.12 Benchmarkresultaten bij het toepassen van verbeteringen uit sectie 4.5.3 voor het Triceratops-model . . . . .	46
4.13 Temporele coherentie bij 6 opeenvolgende frames in het stochastisch algoritme 4.4 . . . . .	47
4.14 Suggestieve contouren-beweging bij radiale transformatie camerapositie	48
4.15 Beweging van een punt op de suggestieve contour . . . . .	49

4.16 Invloed van gecombineerde limietwaarde $t_d$ op het aantal weergegeven contourlijnen . . . . .	51
4.17 Verbetering temporele coherente door het filteren van microcontouren en fading . . . . .	52
4.18 Suggestieve highlights in de comicbook-kunst: Frank Miller, <i>Sin City</i> . . . . .	54
4.19 Lussen van $\kappa_r = 0$ . . . . .	55
4.20 Exagerrated shading met suggestieve highlights . . . . .	56
4.21 Foutieve reliëf impressie bij het gebruik van SC én SH . . . . .	57
4.22 Renders in de stijl van Frank Miller, m.b.v. zwart-witte toon shading . . . . .	58
 5.1 Een moderne GPU pipeline (zonder <i>geometry processing</i> ) . . . . .	60
5.2 Texture Mapping proces . . . . .	63
5.3 Een textuur van 128x128 texels en zijn mipmaps, in aflopende grootte . .	63
5.4 De textuurmap die gebruikt wordt om SC te tekenen via texture mapping	64
5.5 Performantie van het algoritme 5.1 vergeleken met CPU-algoritmes. . . . .	66
5.6 Lijndikte bij texture mapping . . . . .	67
5.7 Invloed van $\kappa_r$ in de limiet voor contouren . . . . .	69
5.8 Invloed van $D_w \kappa_r$ in de limiet voor suggestieve contouren . . . . .	70
5.9 Opbouw van het GPU-shader algoritme om contourlijnen te tekenen . .	71
5.10 Performantie van de GPU-shader implementatie . . . . .	75
5.11 Renders gemaakt met de GPU-shader . . . . .	77
 6.1 De verschillende stappen in een Image Space algoritme. . . . .	79
6.2 Vierkant met uniform verdeelde texture-coördinaten . . . . .	81
6.3 Resultaten van algoritme 6.1, toegepast op het Elephant-testobject met verschillende waarden voor de limiet $t_c$ . . . . .	83
6.4 Effect van <i>flat</i> versus <i>gouraud</i> shading op algoritme 6.1 . . . . .	84
6.5 Het opbouwen van de z-buffer bij het tekenen van meerdere polygonen.	85
6.6 Het toepassen van de Sobel Filter beschreven in sectie 6.3.1 op een afbeelding. . . . .	86
6.7 Het toepassen het Sobel Filter algoritme 6.4 op het Heptoroid-testmodel	90
6.8 Het verdwijnen van interne contourlijnen . . . . .	90
6.9 Detectie radiale filter-algoritme . . . . .	92
6.10 Overzicht van de implementatie van het radiale filter-algoritme voor extractie van contourlijnen. . . . .	93
6.11 Het Cow-testmodel gerenderd met het radiale filter-algoritme 6.6 . . . .	95
6.12 Probleem bij het toepassen van afstandsafhankelijke radiale filter . . . . .	96
 7.1 Resultaten bereikt met de Object Space algoritmes uit hoofdstuk 4 . . .	100
7.2 Performantieresultaten voor alle algoritmes uit hoofdstuk 4 . . . . .	101
7.3 Resultaten bereikt met het Object Space GPU algoritme 5.2 . . . . .	103
7.4 Resultaten bereikt met het Image Space Radiale Filter-algoritme 6.6 . .	104
7.5 Performantieresultaten van de algoritmes uit hoofdstuk 5 en 6 . . . . .	105

# Lijst van tabellen

4.1	Gebruikte 3D-modellen . . . . .	24
4.2	Performantie van algoritme 4.3 . . . . .	36
4.3	Filters uit secties 4.4.1 en 4.4.2 toegepast op benchmark-objecten. . . . .	39
4.4	Gemiddeld percentage vlakken ‘afgekeurd’ door $\kappa_r$ -test en backface culling in algoritme 4.3. . . . .	40
4.5	Percentage van suggestieve contouren gevonden bij verschillende rotatiesnelheden van het Triceratops-model . . . . .	46
4.6	Performantievergelijking bij het uitvoeren van de benchmarks na het toevoegen van suggestieve highlights . . . . .	55
6.1	Resultaten van benchmarks met algoritme 6.1, uitgevoerd op een resolutie van 512x512 pixels. . . . .	83
6.2	Gemiddelde verwerkingstijd (in $10^{-6}$ seconden) van de twee passes in algoritme 6.1. . . . .	83
6.3	Resultaten van benchmarks met Sobel Filter-algoritme 6.4, uitgevoerd op een resolutie van 512 x 512 pixels. . . . .	89
6.4	Resultaten van benchmarks met het Sobel Filter-algoritme 6.4 . . . . .	89
6.5	Gemiddelde framerate (in frames/seconden) bij het uitvoeren van de benchmarks met 6.6 . . . . .	94
6.6	Resultaten van benchmarks met het radiale filter-algoritme 6.6 . . . . .	95

# Lijst van codeblokken

4.1	NdotV Algoritme (Pseudo Code) . . . . .	26
4.2	Face Edges Algoritme (Pseudo Code) . . . . .	26
4.3	Suggestieve contouren op een mesh (Pseudo Code) . . . . .	34
4.4	Markosian Stochastisch Algoritme (Pseudo Code) . . . . .	43
5.1	Suggestieve contouren via Texture Mapping (Pseudo Code) . . . . .	65
5.2	S. Contouren via GPU-shader - CPU gedeelte (Pseudo Code) . . . . .	72
5.3	S. Contouren via GPU-shader - Vertex Shader (GLSL) . . . . .	73
5.4	S. Contouren via GPU-shader - Fragment Shader (GLSL) . . . . .	74
6.1	Contouren via Threshold filter - CPU gedeelte (Pseudo Code) . . . . .	81
6.2	Contouren via Threshold filter - Vertex Shader (GLSL) . . . . .	81
6.3	Contouren via Threshold filter - Fragment Shader (GLSL) . . . . .	82
6.4	Contouren via Sobel filter - CPU gedeelte (Pseudo Code) . . . . .	87
6.5	Contouren via Sobel filter - Fragment Shader (GLSL) . . . . .	88
6.6	S. Contouren via Radiale Filter - Fragment Shader (GLSL) . . . . .	94

# Lijst van afkortingen en symbolen

## Afkortingen

FPS	Frames Per Second
C	(Normale) Contour
SC	Suggestieve Contour
SH	Suggestieve Highlight
CG	Contour Generator
SCG	Suggestieve Contour Generator
GPU	Graphics Processing Unit
CPU	Central Processing Unit
VBO	Vertex Buffer Object
OpenGL	Open Graphics Library

## Symbolen / Benamingen

De volgende symbolen worden consistent gebruikt om geometrische eigenschappen aan te duiden:

$\mathbf{p}$	Willekeurig (vertex)punt op een oppervlak
$\mathbf{n}(\mathbf{p})$	Eenheidsnormaalvector in het punt $\mathbf{p}$
$\mathbf{v}(\mathbf{p})$	View vector in het punt $\mathbf{p}$
$\kappa_r(\mathbf{p})$	Radiale curvatuur in het punt $\mathbf{p}$
$\kappa_1(\mathbf{p}), \kappa_2(\mathbf{p})$	Principale curvaturen in het punt $\mathbf{p}$
$\mathbf{e}_1(\mathbf{p}), \mathbf{e}_2(\mathbf{p})$	Principale curvatuurrichtingen in het punt $\mathbf{p}$
$K(\mathbf{p})$	Gaussiaanse curvatuur in het punt $\mathbf{p}$

# Hoofdstuk 1

## Inleiding

### 1.1 Lijntekeningen

We definiëren lijntekeningen als de visuele stijl waarin een bepaald object wordt afgebeeld door exclusief gebruik te maken van lijnen.

De basiselementen van deze lijntekeningen vormen het onderwerp van deze masterproef: de zogenaamde *contourlijnen*: Door middel van deze lijnen kan in een lijntekening informatie gegeven worden over de vorm, curvatuur en positie van een bepaald object. Deze lijnen kunnen allen dezelfde lijnstijl hebben, of men kan variëren in bijvoorbeeld lijnbreedte, kleur, transparantie ... Een grondige analyse en wiskundige definitie van deze contourlijnen volgt in hoofdstuk 2.

Contourlijnen behoren tot de meest eenvoudige elementen om informatie over een object weer te geven. Deze visuele simplicitet betekent echter niet dat het berekenen en tekenen van deze lijnen een triviale opdracht is, integendeel. In deze tekst worden *computer graphics*-methodes geïmplementeerd en vergeleken om deze contourlijnen te vinden en weer te geven. Het berekende resultaat willen we vervolgens in real-time kunnen manipuleren, dus de performantie van deze methodes is van groot belang.

### 1.2 Computer Graphics

De term *computer graphics* wordt gebruikt voor alle beelden die gemaakt worden met behulp van computers, alsook de verschillende technieken die daarbij worden toegepast. De ontwikkelingen binnen dit domein, een relatief jonge subtak binnen de computerwetenschappen, heeft ervoor gezorgd dat we data eenvoudiger kunnen visualiseren, interpreteren en met elkaar communiceren. Tijdens de laatste vier decennia heeft de evolutie van computer graphics onder meer zijn stempel gedrukt op de entertainmentindustrie: denk aan animatiefilms, special effects en video games. Computer graphics en zijn toepassingen zijn - om het met een cliché te zeggen - niet meer weg te denken uit onze leefwereld.

Een historisch overzicht geven van het met rasse schreden vorderend onderzoek of een bespreking van de belangrijkste technieken die door de jaren heen ontwikkeld werden zou in het kader van deze masterproef te ver leiden. We zullen ons dus

beperken tot de definitie en omkadering van enkele begrippen die in dit werk veelvuldig gebruikt worden.

### 1.2.1 Terminologie

In dit werk worden de volgende termen herhaaldelijk gebruikt:

- **Mesh** (ook **model** of **object** genaamd): indien we met behulp van computer graphics-technieken een beeld willen maken van een bepaald object, hebben we een manier nodig om dit object geometrisch voor te stellen. De meest gebruikte definitie is een *mesh*: dit is een gediscretiseerde vorm van een continue figuur. In de definitie van een mesh wordt een lijst van geometrische punten gegeven met coördinaten  $x_i, y_i$  en  $z_i$ , aangevuld met een lijst van vlakken. Hoe meer punten, hoe gedetailleerder de mesh in kwestie: de effecten van de discretisatie zullen dan minder merkbaar zijn in het uiteindelijke resultaat.
- **Polygon**: Een vlak gedefinieerd in de beschrijving van een mesh wordt een polygon genoemd: een veelhoek. Meestal worden meshes opgebouwd uit driehoeken (*triangles*), maar alle mogelijke veelhoeken zijn mogelijk. In dit werk zullen we met deze *triangulaire* meshes werken.
- **Vertex**: Een punt gedefinieerd in de beschrijving van een mesh wordt een vertex genoemd. Verschillende vertices vormen de hoekpunten van een polygon. Per vertex kunnen er extra eigenschappen gedefinieerd worden. Meestal wordt ook de **normaalvector** gedefinieerd: dit is de vector loodrecht op het vlak rakend aan die vertex. Ook andere eigenschappen zoals bijvoorbeeld de kleur of textuurinformatie kunnen gedefinieerd worden in de vertices. De waarden in de vertices worden lineair geïnterpoleerd over de polygonen.
- **Renderen**: Het proces waarin men, vertrekende van een mesh en een bepaald scenemodel, een 2D-voorstelling (afbeelding) maakt van een bepaalde scène wordt *renderen* genoemd. Het scenemodel bevat informatie over de positie van het oogpunt, de lichtbronnen, het gebruikte lichtmodel, en de materialen waaruit de objecten zijn opgebouwd. Er zijn twee belangrijke methodes om een 2D-beeld te bekomen:
  - **Rasterizing**: Bij het *rasterizen* van een geometrische figuur worden 3D-coördinaten  $x_i, y_i$  en  $z_i$  door middel van matrixtransformaties op een 2D-venster geprojecteerd. Deze manier van renderen is tot op heden de techniek die grafische hardware toepast. In dit werk wordt er (impliciet) ook op deze manier gerenderd.
  - **Raytracing**: Bij het *raytracen* worden vanuit het beschouwde oogpunt lichtstralen geschoten door het 2D-venster (in de 3D-ruimte) naar de scène. Aan de hand van de interactiepunten van deze lichtstralen met de aanwezige geometrie kan de afbeelding gevormd worden. Met deze

techniek is het eenvoudiger om een hoge graad van realisme te bekomen, maar aan een hogere computationele kost.

- **Pixel:** De uiteindelijke afbeelding is gediscretiseerd in pixels: kleine vierkante elementen die 1 welbepaalde kleur kunnen aannemen, meestal gespecificeerd in  $(r, g, b)$ -waarden. De grootte van een bepaalde afbeelding, **resolutie** genaamd, wordt uitgedrukt in pixels.
- **Frame:** In de context van deze masterproef bedoelen we met een *frame* één afbeelding uit een sequentie gegenereerde afbeeldingen. Het menselijk oog heeft gemiddeld 24 frames per seconde nodig om een sequentie te ervaren als vloeiend ‘bewegend’ beeld.

### 1.3 Fotorealistisch versus Niet-Fotorealistisch renderen

Vele van de algoritmes ontwikkeld binnen het domein van computer graphics hebben als doel een reële eigenschap van objecten en hun interactie met de omgeving te modelleren. Denk bijvoorbeeld aan het weergeven van reflectie, schaduwen, breking van licht, ... Deze technieken worden geklasseerd onder de naam *fotorealistisch renderen*. Hun uiteindelijke doel kan geformuleerd worden als: een afbeelding maken die door een menselijke waarnemer niet te onderscheiden valt van (een foto van) de werkelijkheid.

In het gebied van niet-fotorealistisch renderen (kortweg NPR, van Non Photorealistic Rendering) is het doel om afbeeldingen te maken in een bepaalde visuele *stijl*. De term NPR werd voor het eerst gebruikt in [Winkenbach and Salesin, 1996], en hoewel er veel kritiek kwam op het feit dat het erg ‘on-wetenschappelijk’ was om iets te definiëren door vast te leggen wat het niet is blijft de term tot op de dag van vandaag in gebruik. Niet-fotorealistisch renderen wordt nogal vaak gelinkt aan het simuleren van artistieke stijlen (olie op doek, pentekening, cartoon rendering), en tijdens de beginjaren werd het meeste onderzoek ook binnen dit topic uitgevoerd [Meier, 1996]. Het toepassingsgebied van NPR is de laatste jaren echter veel breder geworden.

Vaak wordt niet-fotorealistisch renderen ook toegepast om afbeeldingen van ingewikkelde scenes *eenvoudig* weer te geven, zodat een waarnemer zonder gestoord te worden door *visual overload* kan begrijpen wat er wordt afgebeeld. Verschillende personen interpreteren afbeeldingen echter op verschillende manieren. De moeilijkheid ligt dan ook vaak in het vinden van een visuele stijl waarvan de interpretatie een algemeen karakter heeft, en er vervolgens in slagen om deze stijl mathematisch te definiëren.

De eerder besproken lijntekeningen zijn een vorm van NPR: in deze masterproef wordt een 3D-voorstelling van een bepaald object als beginpunt van de algoritmes verondersteld. Er wordt dus van geometrie in drie dimensies, bekeken vanuit een bepaald camerastandpunt, overgegaan naar een 2D-afbeelding, gerenderd als een lijntkening. Op deze lijntkening willen we door middel van contourlijnen zoveel mogelijk informatie over het beschouwde object weergeven.

## 1.4 Doel

Deze masterproef heeft als doel om:

- De bestaande algoritmes om contourlijnen te berekenen vertrekkende van een 3D-mesh te implementeren, evalueren en vergelijken.
- Op basis van deze analyses nieuwe algoritmes voor te stellen om contourlijnen te extraheren en te tekenen, alsook om deze nieuwe algoritmes efficiënt te implementeren.

De focus van deze masterproef ligt op het berekenen en tekenen van verschillende soorten contourlijnen aan *interactieve* snelheden. We willen 3D-modellen gerenderd met contourlijnen in *real-time* kunnen manipuleren, zonder daarbij te moeten inboeten aan performantie of correctheid.

## Hoofdstuk 2

# Contourlijnen: Situering en Theorie

### 2.1 Contouren en Suggestieve Contouren: intuïtief

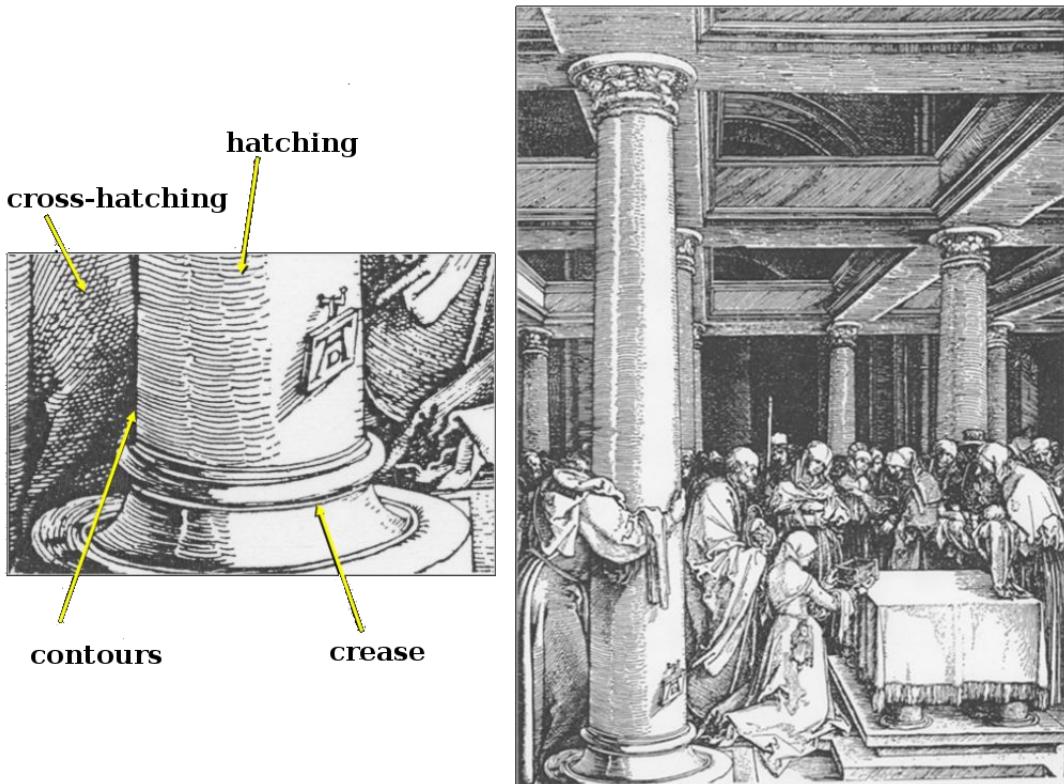
Lijntekeningen brengen een groot aantal lijnen samen om een coherente afbeelding te maken van een bepaalde scène. Een goed voorbeeld van deze techniek vinden we bij de middeleeuwse grootmeester Dürer in figuur 2.1. Verschillende technieken die gebruik maken van lijnen worden hier toegepast: *contouren* en *vouwen* (*creases*) om vorm en rondingen weer te geven, alsook het gebruik van *hatching* en *cross-hatching* (het symmetrisch kruisen van lijnen) om schaduw weer te geven.

Ons visuele systeem slaagt erin om deze afbeelding correct te interpreteren, hoewel de aanwezige lijnen vaak heel verschillende functies hebben: sommige geven schaduw weer, andere duiden geometrie aan. Dit is op zich vrij verrassend: elke lijn kan immers veroorzaakt worden door een oneindig aantal 3D-curves in de ruimte die geprojecteerd worden op het 'venster' van de afbeelding. En toch leidt het brein dus informatie af over de vorm, positie en belichting van het object uit deze lijnen. Een diepgaande neuropsychologische verklaring van dit proces is tot op vandaag nog onbekend, en zou ons binnen de context van dit werk te ver leiden. Er wordt verdergegaan op één belangrijke set van lijnen: *contourlijnen*, het onderwerp van deze masterproef. De twee belangrijkste soorten *contourlijnen* worden in de volgende secties toegelicht: *normale contouren* en *suggestieve contouren*.

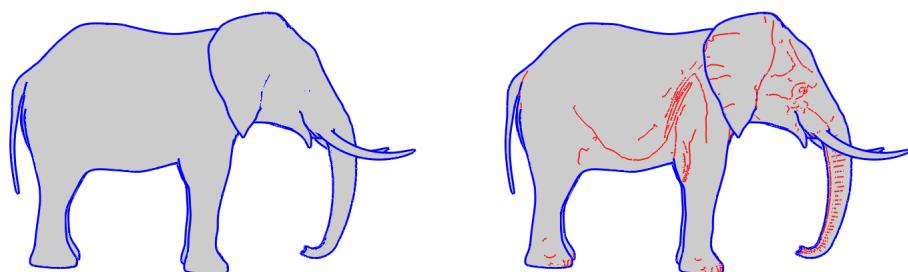
#### 2.1.1 Normale contourlijnen als efficiënte *visual cue*

Normale contourlijnen definiëren we hier als lijnen die we tekenen op de plaatsen waar een oppervlak van een object wegdraait van de toeschouwer en dus bijgevolg onzichtbaar wordt. Een voorbeeld hiervan is weergegeven in figuur 2.2. De blauwe lijnen geven aan waar het oppervlak van de olifant onzichtbaar wordt vanuit het huidige camerastandpunt, dat de positie van ons oog tegenover het object voorstelt.

Op het visuele niveau zijn contouren een *primaire* aanwijzing voor ons objectherkenningsysteem [Koenderink, 1984]. Als het menselijke visuele verwerkingsysteem



Figuur 2.1: Lijntekening Kunsthalle, A. Dürer (1505). Met behulp van verschillende lijntekniken kunnen effecten bereikt worden die informatie geven over schaduw, diepte en curvatuur van de afgebeelde objecten.



Figuur 2.2: Render van het Olifant-model met gewone contouren en suggestieve contouren. De suggestieve contouren brengen meer detail aan in de figuur. Dit is bijvoorbeeld goed te zien in de slurf van het model.

een waargenomen object probeert te identificeren spelen contourlijnen een belangrijke rol in deze verwerking. Bepaalde theorieën uit de neuropsychologie beweren dat een waargenomen contourlijn als het ware *gematcht* wordt met een interne ‘database’ van reeds waargenomen (en dus geklassificeerde) contourlijnen. Biederman haalde het belang van contouren voor patroonherkenning aan in het blad *Psychological Review* [Biederman, 1987]. Door middel van experimenten onderzochten hij en zijn team hoe de herkenningsgraad van bepaalde objecten door een testgroep exponentieel afnam bij het verbergen van een bepaald percentage contourlijnen.

Een andere, eerder intuïtieve, aanwijzing die het belang van contourlijnen aanduidt is terug te vinden in het tekenproces van kleuters: kinderen beginnen vaak met het tekenen van contouren als hen gevraagd wordt om alledaagse objecten te illustreren [Deregowski and Dziurawiec, 1996].

Hieruit kan ook verklaard worden waarom de meeste artiesten en kunstenaars bij het weergeven van diverse objecten in hun werk veel aandacht besteden aan deze contourlijnen. Er zijn duizenden verschillende attributen en eigenschappen die in rekening kunnen gebracht worden om een bepaald object weer te geven, maar vaak wil een artiest/ontwerper alleen die attributen behouden die een hoge *informatiewaarde* hebben: met behulp van deze attributen kan de toeschouwer/gebruiker snel en efficiënt een bepaald object herkennen. Een goed voorbeeld hiervan zijn handleidingen en instructies: men wil de gebruiker hier immers snel en duidelijk informatie leveren, en niet verwarringen met overbodige details. Men zou zelfs verder kunnen gaan door te stellen dat het snel herkennen en begrijpen van objecten op noodinstructies (bvb: pamflet met instructies voor vliegtuigcrash) van levensbelang kan zijn.

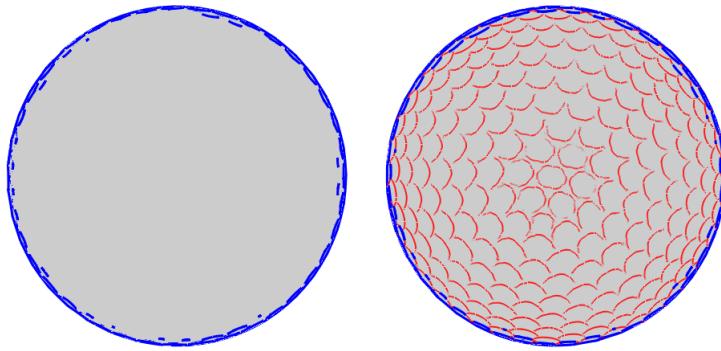
Waarom een artiest bepaalde lijnen al dan niet selecteert ter opneming in een figuur is moeilijk in een definitie te gieten - het subjectieve en de eigen visie op esthetiek spelen hierin vaak een rol. Wel staat vast dat contourlijnen de absolute basis vormen om een object weer te geven: ze scheiden het object van de rest van de omgeving, zodat het als individueel element kan beschouwd worden in een scène.

In de resterende tekst van deze masterproef zal het gebruik van de term ’*contourlijnen*’ zonder specificatie altijd duiden op de hier besproken normale contourlijnen.

### 2.1.2 Suggestieve contourlijnen als noodzakelijke aanvulling

Vaak volstaan *enkel* contourlijnen echter niet om een duidelijke voorstelling van een bepaald object te maken, zoals te zien is op figuur 2.3. Hiervoor wordt een uitbreiding ingevoerd: *suggestieve contourlijnen*. Deze kunnen intuïtief beschreven worden als *bijna*-contouren: het zijn lijnen die locaties accentueren waar in een nabijgelegen camerastandpunt een gewone contour zal ontstaan: contouren en suggestieve contouren vloeien dus in elkaar over als het camerastandpunt zich rond het object verplaatst. Zo vormen suggestieve contouren een natuurlijke aanvulling op normale contourlijnen, zoals verder zal bewezen worden.

Suggestieve contouren kunnen extra *detail* toevoegen aan figuren. Zo krijgt men door toevoeging van suggestieve contouren in figuur 2.2 veel extra informatie over de figuur: o.a. het verdere verloop van het oor van de olifant, de huidbanden in de slurf en de locatie van het oog. Hiermee wordt ook een tweede belangrijke functie van



*Figuur 2.3: Golfbal: gewone contouren en suggestieve contouren. Zonder suggestieve contouren zou dit model niet herkenbaar zijn als een golfbal: er is belangrijke informatie aanwezig in de curvatuur die door de suggestieve contouren aangegeven wordt.*

suggestieve contouren duidelijk: informatie verstrekken over de *curvatuur* van het beschouwde object: met behulp van de extra lijnen kan afgeleid worden waar er zich plooien bevinden in de huid van de olifant.

De noodzaak van suggestieve contouren komt nog meer tot uiting wanneer het een object betreft waar essentiële informatie voor de herkenning net ligt in die eigenschappen van het model die door suggestieve contouren geaccentueerd worden. Zonder deze suggestieve contouren zou het in figuur 2.3 niet duidelijk zijn dat het object een golfbal betreft, omdat alle nodige details voor een correcte herkenning (de cirkelvormige inkepingen) zich bevinden in deze lijnen.

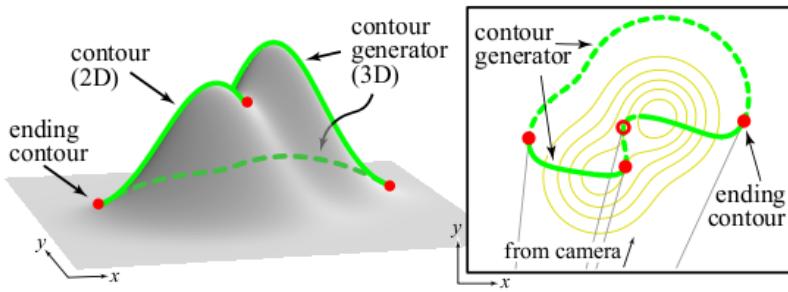
## 2.2 Contouren: definitie

Om normale contouren mathematisch te definiëren wordt gebruik gemaakt van een *Contour Generator*. Deze mathematische constructie, voor het eerst ingevoerd door [Cipolla and Giblin, 2000], beschrijft de set punten op een oppervlak die, bekeken vanuit een bepaald camerastandpunt, deel uitmaken van een contour. Deze generator wordt beschreven met de formule:

$$\mathbf{n}(\mathbf{p}) \cdot \mathbf{v}(\mathbf{p}) = 0 \quad (2.1)$$

waarin  $\mathbf{p}$  de positie is van een punt op het oppervlak van het object,  $\mathbf{n}(\mathbf{p})$  de genormaliseerde normaalvector in  $\mathbf{p}$  en  $\mathbf{v}$  de *view vector*, berekend als het verschil tussen de positie van de camera  $\mathbf{c}$  en de positie van het beschouwde punt:  $\mathbf{v}(\mathbf{p}) = \mathbf{c} - \mathbf{p}$ .

In [Decarlo et al., 2003] wordt beschreven hoe de Contour Generator een reeks onverbonden lussen op het oppervlak van het object voorstelt. De *contour* zelf wordt bekomen als men de zichtbare stukken neemt van deze lussen en deze projecteert op een vlak. Wanneer de Contour Generator dus bekeken wordt vanop een loodrecht vlak ziet men geen lussen, maar lijnen die abrupt stoppen: dit zijn de contourlijnen



Figuur 2.4: **Links:** De Contour Generator. **Rechts:** Indien de Contour Generator bekeken wordt vanuit een bepaald camerastandpunt zijn aparte, onderbroken contourlijnen zichtbaar. (Figuur uit [Decarlo et al., 2003])

die te zien zijn vanuit een bepaald camerastandpunt. In figuur 2.4 worden de Contour Generator en de projectie van de contourlussen geïllustreerd.

Indien men de contouren wilt tekenen van een object dat gediscretiseerd werd in polygonen kan de Contour Generator eenvoudiger gedefinieerd worden: het is dan de set van lijnen die tussen twee polygonen liggen, waarvan slechts één polygoon zichtbaar is vanuit het huidige camerastandpunt [Appel, 1967]. Voor oppervlakken die niet gediscretiseerd werden zal echter de functie (2.1) voor elk punt op het oppervlak moeten geëvalueerd worden.

## 2.3 Suggestieve contouren: definitie

Na de eerder intuïtieve situering van suggestieve contouren als uitbreiding op normale contourlijnen in sectie 2.1.2 worden in deze sectie enkele formele definities opgesteld.

### 2.3.1 Radiale curvatuur

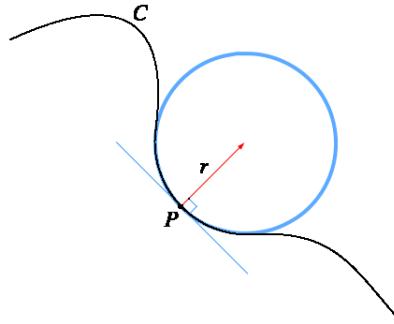
Om suggestieve contouren te beschrijven is het begrip *radiale curvatuur* vereist.

De *curvatuur*  $\kappa_C(\mathbf{p})$  van een curve  $C$  in een punt  $\mathbf{p}$  kan men definiëren als de inverse van de straal van de cirkel die aansluit bij de curve in het punt  $\mathbf{p}$  [Hilbert and Cohn-Vossen, 1937].

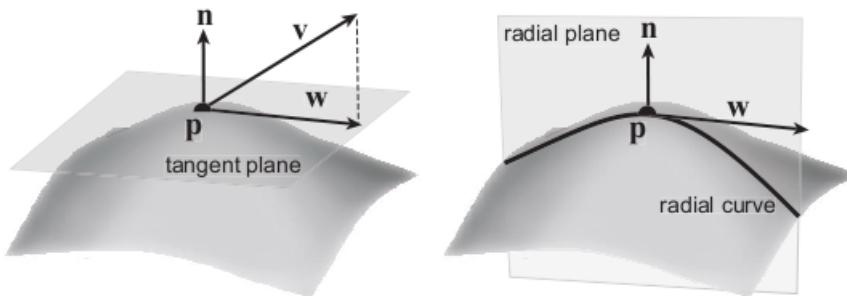
$$\kappa_C(\mathbf{p}) = \frac{1}{r} \quad (2.2)$$

Dit wordt geïllustreerd in figuur 2.5. Hoe kleiner de kromming van de curve, hoe groter de straal vereist om een aansluitende cirkel te construeren, en bijgevolg hoe kleiner de curvatuur.

De curvatuur  $\kappa_S(\mathbf{p})$  van een oppervlak  $S$  in een punt  $\mathbf{p}$  kan men definiëren via een gekozen curve die op het oppervlak ligt en het punt  $\mathbf{p}$  bevat [DoCarmo, 1976]. Deze curve wordt bekomen door  $S$  te snijden met het vlak bepaald door het punt  $\mathbf{p}$ , de eenheidsnormaalvector  $\mathbf{n}$  en een gekozen richting  $\mathbf{d}$ . Deze richting  $\mathbf{d}$  moet in het vlak liggen dat  $S$  loodrecht raakt in het punt  $\mathbf{p}$ . De richtingen waarin deze



Figuur 2.5: De curvatuur van een curve  $C$  in een punt  $p$  wordt bepaald als de inverse van de straal van de best aansluitende cirkel.



Figuur 2.6: **Links:** Definitie van de richtingsvector  $w$  door projectie van de view vector  $v$  op een vlak rakend aan het oppervlak in het punt  $p$ . **Rechts:** De curve gebruikt voor het bepalen van de radiale curvatuur  $\kappa_r(p)$ . (Figuur uit [Decarlo et al., 2003])

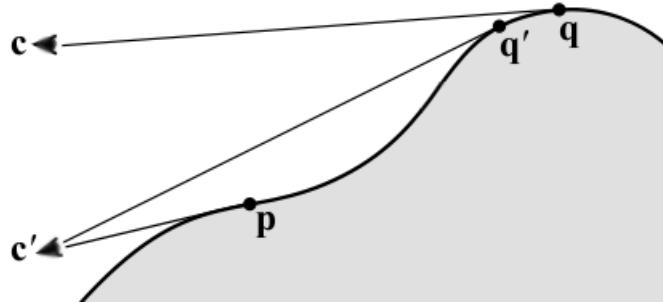
curvatuur respectievelijk zijn maximum en minimum bereikt noemen we de *principale curvatuurrichtingen* van het oppervlak  $S$  in het punt  $p$ . De curvatuurwaarden in deze twee richtingen worden aangeduid door  $\kappa_1(p)$  en  $\kappa_2(p)$  en worden de *principale curvatuuren* genoemd.

Belangrijk voor de definitie van suggestieve contouren is de curvatuur in het punt  $p$  in de richting  $w$ , waarbij  $w$  bekomen wordt door de view vector  $v$  te projecteren op het loodrechte vlak rakend aan het oppervlak  $S$  in het punt  $p$  (zie afbeelding 2.6). Deze curvatuur is de *radiale curvatuur*  $\kappa_r(p)$  [Koenderink, 1984].

### 2.3.2 Suggestieve contouren op oppervlakken

De verdere uitwerking van definities voor suggestieve contouren is gebaseerd op [Decarlo et al., 2003], de referentiepaper aangaande suggestieve contouren. Nagenoeg alle verdere academisch werk aangaande suggestieve contouren bouwt verder op deze paper.

Zoals eerder vermeld in sectie 2.1.2 kunnen we suggestieve contouren informeel beschrijven als *bijna*-contouren: het zijn de locaties op een oppervlak die contouren



Figuur 2.7: Punt  $p$  is onderdeel van de suggestieve contour gezien vanuit standpunt  $c$ , punt  $q'$  niet. (Figuur gebaseerd op [Decarlo et al., 2003])

zullen worden wanneer men het camerastandpunt aanpast naar een andere, dichtbijzijnde positie. In figuur 2.7 wordt duidelijk gemaakt welke punten op een oppervlak in aanmerking komen als onderdeel van een suggestieve contour:

Beschouw twee camerastandpunten,  $c$  en  $c'$ . In het camerastandpunt  $c$  is punt  $q$  onderdeel van een normale contour ( $\mathbf{n}(q) \cdot \mathbf{v}(q)$  is daar immers 0). Als de camera wordt bewogen naar positie  $c'$  zal de contour in punt  $q$  naar beneden ‘glijden’ langs verschillende punten op het oppervlak, zoals bijvoorbeeld in punt  $q'$ : dit punt maakt geen deel uit van een suggestieve contour in camerastandpunt  $c$ , omdat het punt  $q'$  corresponderende normale contourlijnen heeft in nabije camerastandpunten. Punt  $p$  daarentegen, zal pas in camerastandpunt  $c'$  plots verschijnen: het maakt deel uit van een suggestieve contour in camerastandpunt  $c$ . Suggestieve contouren zijn dus reeksen van punten waar het oppervlak een radiale curvatuur heeft die gelijk is aan 0 en het oppervlak bovendien (convex) wegbuigt van de camera.

### 2.3.3 Suggestieve Contour Generator

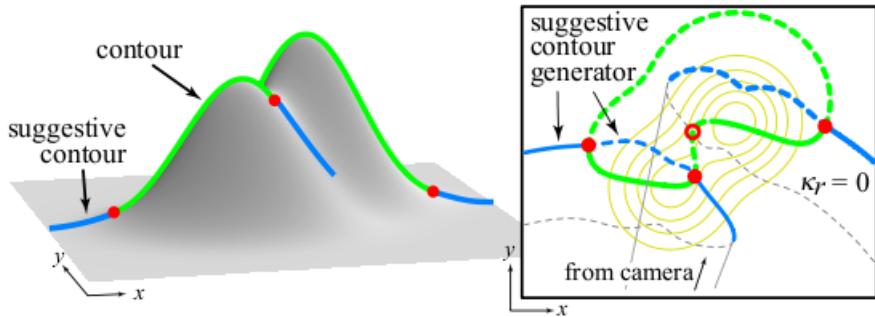
Net zoals de generator voor normale contouren wordt nu een Suggestieve Contour Generator gedefinieerd: het is de set van punten op het oppervlak  $S$  waar de radiale curvatuur  $\kappa_r$  gelijk is aan 0, en de richtingsafgeleide van  $\kappa_r$  in de richting van  $\mathbf{w}$  positief is. Formeel:

$$\kappa_r = 0 \text{ en } D_{\mathbf{w}}\kappa_r = 0 \quad (2.3)$$

Ook deze generator vormt lussen op het oppervlak  $S$ , waarvan de zichtbare delen na projectie vanuit het camerastandpunt de suggestieve contourlijnen voorstellen. Aangezien de eindpunten van normale contouren ook een radiale curvatuur van 0 hebben (de vector  $\mathbf{n}$  staat daar immers loodrecht op  $\mathbf{v}$ ) vloeien de suggestieve contouren over in normale contouren en vice versa [Koenderink, 1984].

### 2.3.4 Alternatieve definitie: *bijna*-contouren

In de volgende secties worden ook enkele alternatieve definities van de Suggestieve Contour Generator opgenomen: deze zullen immers de basis vormen voor de ver-



Figuur 2.8: **Links:** De Suggestieve Contour Generator. **Rechts:** Suggestieve contouren vloeien over in gewone contouren, en vice versa. (Figuur uit [Decarlo et al., 2003])

schillende technieken om contouren en suggestieve contouren te berekenen en weer te geven.

Zoals werd aangegeven in 2.1.2 kunnen we suggestieve contouren ook intuïtief beschrijven als *bijna*-contouren: het zijn posities waarvoor de Contour Generator (2.1) niet 0 is, maar een lokaal minimum bereikt in de richting van  $\mathbf{w}$ . Dit is equivalent aan eisen dat de richtingsafgeleide van  $\mathbf{n} \cdot \mathbf{v}$  in de richting van  $\mathbf{w}$  0 is:

$$D_{\mathbf{w}}(\mathbf{n} \cdot \mathbf{v}) = 0, \text{ en} \quad (2.4)$$

alsook dat de tweede afgeleide in de zelfde richting groter dan 0 is, wat duidt op een lokaal minimum:

$$D_{\mathbf{w}}(D_{\mathbf{w}}(\mathbf{n} \cdot \mathbf{v})) > 0 \quad (2.5)$$

In [Decarlo et al., 2003] wordt aangetoond dat we (2.4) op de volgende manier kunnen herschrijven om equivalentie met (2.3) te bewijzen:

$$D_{\mathbf{w}}(\mathbf{n} \cdot \mathbf{v}) = D_{\mathbf{w}}\mathbf{n} \cdot \mathbf{v} + \mathbf{n} \cdot D_{\mathbf{w}}\mathbf{v} \quad (2.6)$$

waarin we in de eerste term  $\mathbf{v}$  kunnen vervangen door  $\mathbf{w}$  (gezien  $\mathbf{n}$  de eenheidsnormaalvector is). Afgeleiden van  $\mathbf{v}$  liggen in het rakend vlak aan het oppervlak  $S$  in  $\mathbf{p}$ , dus in de tweede term staat  $D_{\mathbf{w}}\mathbf{v}$  loodrecht op  $\mathbf{n}$ , waardoor het product in deze tweede term 0 wordt. Bijgevolg kunnen we (2.6) schrijven als:

$$D_{\mathbf{w}}(\mathbf{n} \cdot \mathbf{v}) = D_{\mathbf{w}}\mathbf{n} \cdot \mathbf{w} = -II(\mathbf{w}, \mathbf{w}) = (\mathbf{w}, \mathbf{w})\kappa_r \quad (2.7)$$

waarin  $II$  staat voor de zogenaamde *tweede fundamentele vorm*, ook wel de *shape tensor* genoemd. Deze functie werd geïntroduceerd in [DoCarmo, 1976] en [Guggenheim, 1977] en is een kwadratische functie met dezelfde nulpunten als  $\kappa_r$ . Hiermee is bewezen dat de formules (2.4) en (2.5) equivalent zijn aan de eerste definitie van de Suggestieve Contour Generator (2.3).

### 2.3.5 Alternatieve definitie: Contouren in nabije camerastandpunten

Een derde en laatste definitie van de Suggestieve Contour Generator is gebaseerd op het voorbeeld uit figuur 2.7. In sectie 2.3.2 werd gesteld dat punt  $\mathbf{q}'$  daar niet tot een suggestieve contour behoorde in camerastandpunt  $\mathbf{c}$ , omdat punt  $\mathbf{q}'$  *corresponderende* contouren had in nabije camerastandpunten. Om deze informele definitie om te vormen naar een formele definitie van de Suggestieve Contour Generator is een maatstaf vereist om ‘nabijheid’ te kunnen definiëren.

De *radiale afstand* op een punt  $\mathbf{p}$  van camerastandpunt  $\mathbf{c}$  tot een ander camerastandpunt  $\mathbf{c}'$  is de hoek gevormd door de punten  $\mathbf{c}$ ,  $\mathbf{p}$  en de projectie van het camerastandpunt  $\mathbf{c}'$  op het radiale vlak van  $\mathbf{c}$ . Het radiale vlak is gedefinieerd zoals in figuur 2.6: het is het vlak gevormd door het punt  $\mathbf{p}$ , de eenheidsnormaalvector  $\mathbf{n}$  en de richtingsvector  $\mathbf{w}$ , die ook gebruikt wordt in de richtingsafgeleide.

Met behulp van deze radiale afstand kunnen we de derde definitie van de Radiale Contour Generator opstellen [Decarlo et al., 2003]. Merk op dat hierbij ook gebruik gemaakt wordt van de (normale) Contour Generator (2.1), wat nogmaals het impliciete verband onderstreept tussen contouren en suggestieve contouren:

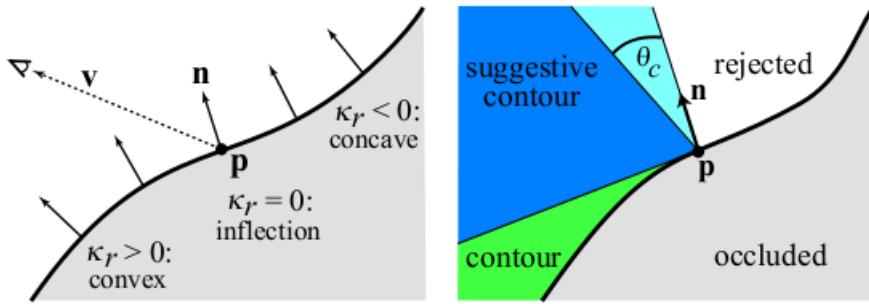
**Definitie.** *De Suggestieve Contour Generator is de set van punten op de Contour Generator van een nabij camerastandpunt (met een radiale afstand van minder dan 90 graden) die niet in radiale correspondentie zijn met punten op Contour Generatoren van enig ander radiaal dichter camerastandpunt.*

In figuur 2.9 is het duidelijk dat het punt  $\mathbf{p}$  een lokaal minimum vormt van  $\mathbf{n} \cdot \mathbf{v}$ . Hierdoor is het volgens de tweede definitie van de Suggestieve Contour Generator (formules (2.4) en (2.5)) reeds een onderdeel van de suggestieve contour.

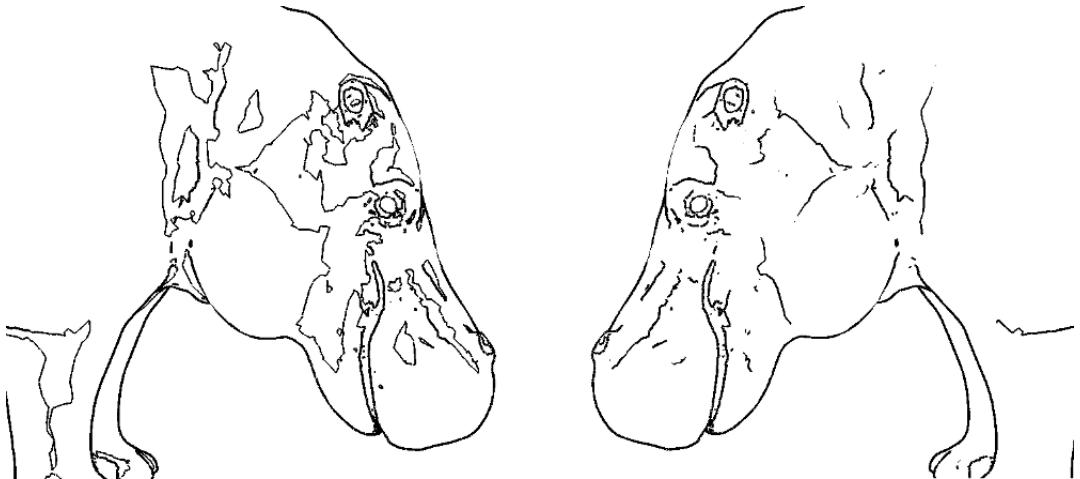
Dit wordt ook aangetoond met de derde definitie van de Suggestieve Contour Generator: Als we het camerastandpunt naar linksonder verplaatsen, zal op het punt  $\mathbf{p}$  een echte contour verschijnen. Dit gebeurt pas wanneer we ons onder het raakvlak aan het oppervlak  $S$  bevinden, aangeduid door de groene zone in figuur 2.9. Het punt  $\mathbf{p}$  is wel in radiale correspondentie met de punten links van zich, maar alleen in Contour Generatoren die horen bij camerastandpunten die een radiale afstand groter hebben dan de afstand tot het camerastandpunt waarop er een contour verschijnt in  $\mathbf{p}$ . Bijgevolg is ook volgens de derde definitie van de Suggestieve Contour Generator het punt  $\mathbf{p}$  een onderdeel van de suggestieve contourlijn.

### 2.3.6 Belang van de $D_{\mathbf{w}\kappa_r}$ test

Zoals weergegeven wordt in figuur 2.9 zal op punt  $\mathbf{p}$  enkel een suggestieve contour zichtbaar zijn wanneer het camerastandpunt zich bevindt in de blauwe zone aangeduid op de figuur. Wanneer de camera in een radiale hoek voorbij de eenheidsnormaalvector  $\mathbf{v}$  wordt geplaatst behoort het punt  $\mathbf{p}$  niet meer tot de set punten bepaald door de Suggestieve Contour Generator: als we de tweede afgeleide maken in de richting van  $\mathbf{w}$  (de neergeslagen *view vector*, zie sectie 2.3.1), is deze negatief: het oppervlak wordt hier concaaf tegenover het camerastandpunt.



Figuur 2.9: **Links:** Punt  $p$  behoort tot de suggestieve contour bekeken vanuit het huidige camerastandpunt. **Rechts:** De limiet  $\theta_c$  wordt ingevoerd om numerieke instabiliteit dichtbij een loodrecht camerastandpunt tegen te gaan: punten van contourlijnen die gevonden werden in het lichtblauwe gebied worden hierdoor genegeerd. (Figuur uit [Decarlo et al., 2003])



Figuur 2.10: Het belang van de  $D_{w\kappa_r}$  test: **Links** zonder test, **Rechts** met deze test. We zien hoe zonder de test op  $D_{w\kappa_r}$  de volledige lussen van radiale curvatuur worden weergegeven: dit komt niet overeen met hoe een artiest/ontwerper het object zou tekenen.

Deze  $D_{w\kappa_r}$  test is heel belangrijk, omdat de lijnen bepaald door  $\kappa_r = 0$  slechts resulteren in lussen op het oppervlak: dit is niet hoe een artiest/ontwerper het object zou weergeven. Het verschil wordt aangetoond op figuur 2.10. We zien hoe met de  $D_{w\kappa_r}$  test alleen de segmenten van deze lussen worden bijgehouden waar het oppervlak convex is t.o.v. de camera.

### 2.3.7 Stabiliteit van de Suggestieve Contour Generator

Als het camerastandpunt een loodrechte positie parallel met de eenheidsnormaalvector  $\mathbf{n}$  in het punt  $\mathbf{p}$  nadert, kan de vector  $\mathbf{w}$  die gebruikt wordt in de tweede afgeleide  $D_{\mathbf{w}}\kappa_r$  snel van richting veranderen: een slechts in kleine mate veranderende view vector  $\mathbf{v}$  wordt immers neergeslagen op het vlak loodrecht rakend in  $\mathbf{p}$ . Indien het camerastandpunt zich dus bevindt in de lichtblauwe zone aangeduid op figuur 2.9 zijn de segmenten van de Suggestieve Contour Generator op die positie zo onstabiel dat ze weinig nuttige informatie kunnen leveren over de vorm van het object. In [Decarlo et al., 2003] wordt dus voorgesteld om een extra constraint in te voeren op punten van de Suggestieve Contour Generator:

$$0 < \theta_c < \cos^{-1}\left(\frac{\mathbf{n}(\mathbf{p}) \cdot \mathbf{v}(\mathbf{p})}{\|\mathbf{v}(\mathbf{p})\|}\right) \quad (2.8)$$

Met behulp van deze beperking kan men contourlijnsegmenten weigeren die onder het gegeven camerastandpunt onstabiel zijn.

## 2.4 Besluit

In dit hoofdstuk werden contourlijnen gesitueerd binnen een algemeen neuropsychologisch en wetenschappelijk kader. Vervolgens werden de zogenaamde *generatoren* beschreven voor contourlijnen en suggestieve contourlijnen: deze mathematische constructies laten toe om de set punten te definiëren op een oppervlak die deel uitmaken van de gewenste lijnen. Voor normale contourlijnen werd de Contour Generator 2.1 opgesteld, voor suggestieve contouren de Suggestieve Contour Generator 2.3, waarvoor ook 2 alternatieve definities gegeven werden in secties 2.3.4 en 2.3.5. Het belangrijke begrip *Radiale Curvatuur* werd gedefinieerd in sectie 2.3.1.

Deze mathematische definities vormen de basis voor de rest van dit werk: hierop zullen de methodes die leiden tot de verschillende manieren om contourlijnen te berekenen en renderen in real-time gebaseerd worden: het zijn de eisen waaraan de getekende lijnen moeten voldoen om *correct* te zijn. Verder zal aangetoond worden dat we deze correctheidseis kunnen verzwakken om aan performantie te winnen, zonder dat daarbij significante visuele informatie verloren gaat. Wel is het zo dat eender welke afgeleide methode rekening zal moeten houden met de stabiliteit van de door de generatoren opgelegde constraints, zoals reeds kort aangehaald in sectie 2.3.7.

## Hoofdstuk 3

# Methodes voor Contourlijnextractie

In secties 2.2 en 2.3 werden definities afgeleid voor de Contour Generator en de Suggestieve Contour Generator. In deze sectie wordt het kader geschetst waarin de methodes die zullen worden behandeld in dit werk vergeleken kunnen worden.

### 3.1 Object Space en Image Space algoritmes

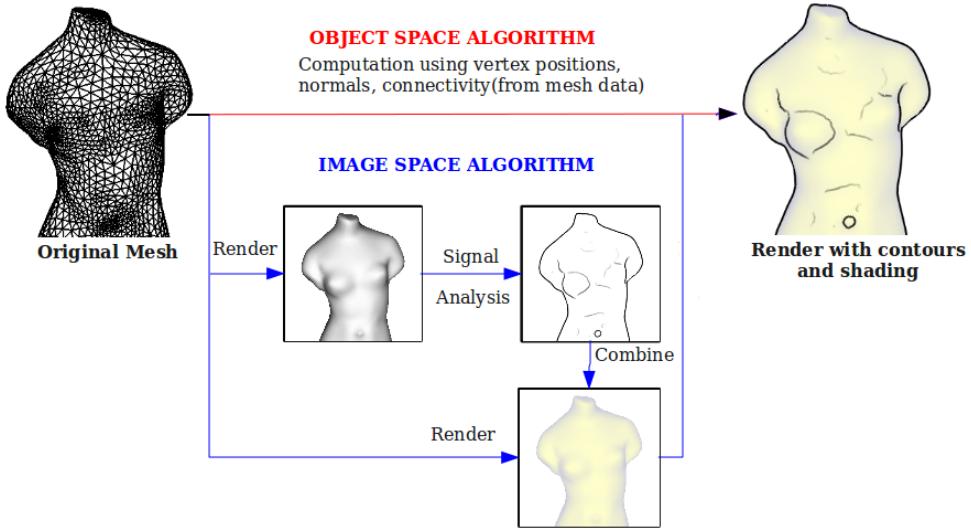
Om de gebruikte methoden te kunnen classificeren moet een onderscheid gemaakt worden tussen zogenaamde *Object Space* algoritmes en *Image Space* algoritmes [Rusinkiewicz, 2008].

- **Object Space algoritmes** worden gedefinieerd als methodes die werken op de originele *mesh* en bijhorende data, zoals bijvoorbeeld posities van vertices, normalen en connectiviteit tussen vertices en vlakken. De beschikbare informatie hangt af van het dataformaat van de *mesh*. Deze methodes werken meestal in het originele assenstelsel van de *mesh*. In de context van contourlijnen tekenen betekent dit dus dat de contourlijnen op de *mesh* zelf zullen worden berekend en gerenderd, gebruik makend van driedimensionale coördinaten.

Object Space algoritmes worden in deze masterproef behandeld in hoofdstukken 4 en 5.

- **Image Space** worden gedefinieerd als methodes die werken op een scalair veld, vaak gevisualiseerd als een afbeelding. Deze methodes werken meestal in het assenstelsel van dit scalair veld.

In de context van contourlijnen betekent dit dat de *mesh* eerst zal gevisualiseerd worden op een bepaalde manier, en op deze visualisatie vervolgens een vorm van signaalverwerking zal worden toegepast om de contourlijnen te extraheren. Deze contourlijnen kunnen dan achteraf over een bepaalde voorstelling van het originele model, bekeken vanuit hetzelfde camerastandpunt, getekend worden om een gecombineerd beeld te vormen.



Figuur 3.1: Object Space / Image Space algoritmes: schema. Bij Object Space algoritmes worden de contourlijnen meteen uit de originele mesh-data geëxtraheerd. Bij Image Space algoritmes wordt eerst een tussentijdse render gemaakt waarop signaalverwerking wordt uitgevoerd om de contourlijnen te extraheren. Vervolgens worden deze gevonden contourlijnen samengevoegd (door convolutie) met een gewenste render om tot het eindresultaat te komen.

Image Space algoritmes worden in deze masterproef behandeld in hoofdstuk 6.

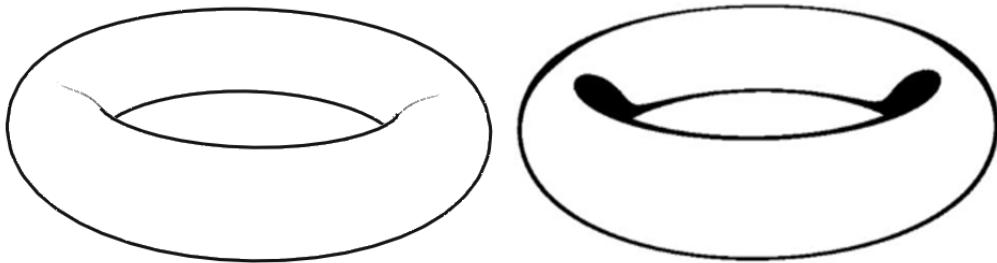
Figuur 3.1 illustreert de veralgemeende werkwijze van de twee soorten algoritmes met een voorbeeld. Het Object Space algoritme werkt meteen op de mesh, waar het Image Space algoritme eerst een diffuse rendering maakt van de mesh om hieruit door middel van signaalanalyse contouren te extraheren. Vervolgens wordt het object nogmaals gerenderd in de gewenste shading en worden de bekomen resultaten samengevoegd.

## 3.2 Eigenschappen

Op enkele punten worden de voordelen en nadelen van Object Space/Image Space algoritmes voor contourlijnextractie duidelijk. In de volgende secties worden deze punten elk afzonderlijk uitgelicht.

### 3.2.1 Controle over lijnstijl

In Object Space algoritmes blijft het concept van een *lijnstuk* behouden: de resultaten van deze algoritmes zijn een reeks segmenten met elk een beginpunt  $(x_i, y_i, z_i)$  en een eindpunt  $(x_j, y_j, z_j)$ , opgesteld in hetzelfde assenstelsel als de originele mesh. Men kan dus voor elk van deze segmenten beslissen in welke stijl deze getekend moeten



*Figuur 3.2: Links: Object Space algoritme. Rechts: Image Space algoritme: minder controle over lijndikte. We verliezen deze controle omdat bij Image Space algoritmes impliciet op een scalair veld van pixels gewerkt wordt, zodat geen lijnsegmenten kan worden opgesteld: er is in het scalaire veld geen geometrische relatie met het originele model aanwezig.*

worden, bijvoorbeeld qua kleur en dikte. Ook kunnen statistieken gegenereerd worden over het aantal segmenten, hun onderlinge afstand, hun gemiddelde lengte, ...

In Image Space algoritmes verliest men deze controle over de lijnsegmenten: aangezien er hier gewerkt wordt op een gerenderd beeld van de mesh kan men geen afzonderlijke segmenten meer definiëren die enige geometrische relatie hebben met de originele mesh. Een bepaald punt  $(x_i, y_i)$  in het assenstelsel van het gerenderd scalair veld zal nu al dan niet tot de contour behoren: dit is de enige informatie waarover men beschikt als resultaat van een Image Space algoritme.

Een voorbeeld van het effect is te zien op figuur 3.2. Aangezien in het Image Space algoritme uit dit voorbeeld slechts een eenvoudige threshold filter gebruikt werd, is de contourlijn ‘uitgesmeerd’ over gebieden met een lage curvatuur. Dit effect is niet zo triviaal te verhelpen: er is immers geen manier om de dikte van een lijnstuk vast te leggen zonder het concept ‘lijnstuk’. Zoals verder zal aangetoond worden zijn problemen in Image Space gerelateerd aan de stijl van de getekende lijnen (gedeeltelijk) te verhelpen door complexere signaalfilter-methoden, gecombineerd met de juiste shading-technieken. Toch blijft dit een fundamenteel en belangrijk verschil tussen de twee klassen contourlijnextractie-algoritmes.

Voor het gebruik van contourlijnen en suggestieve contourlijnen in artistieke toepassingen is controle over de lijnstijl zeer belangrijk: over dit onderwerp zijn al verschillende onderzoeken gebeurd, zoals bijvoorbeeld [Kalnins et al., 2002]. Hierin wordt een editor ontwikkeld die op interactieve wijze een artiest/ontwerper toelaat in enkele camerastandpunten de contourlijnen in een bepaalde stijl te tekenen. Vervolgens wordt deze stijl door aanpassing van de renderparameters zo goed mogelijk gevuld in de andere mogelijke camerastandpunten.

### 3.2.2 Efficiëntie

Aangezien Image Space algoritmes voor contourlijnextractie een algemene aanpak nastreven maken ze weinig of geen gebruik van de computationeel ingewikkeldere elementen uit de definitie van de contourgeneratoren (de tweede richtingsafgeleide  $D_{\mathbf{w}} \kappa_r$ ,

bijvoorbeeld). Ze werken op een specifieke 2D voorstelling (render) van het object, geïllustreerd in figuur 3.1. Deze voorstelling kan in een specifieke implementatie zo gekozen worden dat ze een goede input vormt voor de gebruikte signaalanalyse-techniek, zoals bijvoorbeeld thresholding of randdetectie. Hierdoor zijn deze methodes vaak eenvoudiger dan hun tegenhangers in Object Space, maar gaat dit uiteraard ten koste van de correctheid en controleerbaarheid van het uiteindelijke resultaat.

Ook is het zo dat Image Space algoritmes impliciet uit twee *passes* bestaan, waar bij Object Space algoritmes één pass voldoende is om de contourlijnen te extraheren uit de mesh-data. Dit betekent voor Object Space algoritmes wel dat het aantal mesh-datapunten dat moet geëvalueerd worden een invloed heeft op de complexiteit van het algoritme: de meeste Object Space algoritmes hebben een complexiteit van minstens  $O(n)$ , waarin  $n$  het aantal vertices in de mesh voorstelt.

### 3.2.3 Hardware-ondersteuning

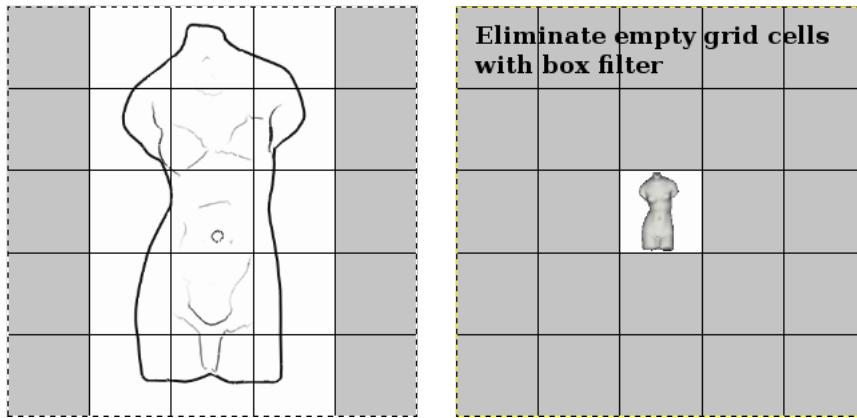
Tot voor kort waren enkel Image Space algoritmes geschikt om hardware-ondersteuning te gebruiken, gezien de standaard stappen van deze algoritmes (zie figuur 3.1) reeds lange tijd in nagenoeg alle grafische hardware ondersteund worden. Het samenvoegen van twee textures (scalaire velden) om tot een samengesteld beeld te komen is bijvoorbeeld een eenvoudige convolutie. Object Space algoritmes kampen daarentegen met andere problemen: zij moeten de mesh-data en andere grote datastructuren bewaren in het actieve geheugen en de vertex en matrix-operaties efficiënt implementeren zonder enige vorm hardware-ondersteuning. Ook moet er manueel gezorgd worden voor parallelisme, een werkwijze die in moderne grafische pipeline impliciet verweven zit. Op deze verschillen wordt in detail ingegaan in hoofdstuk 6.

Met de komst van programmeerbare *vertex shaders* en *fragment shaders*, bijhorende high-level shaderprogrammeertalen en compatibele hardware wordt het echter ook mogelijk voor Object Space algoritmes om vertex-gerelateerde berekeningen en de uiteindelijke rendering (gedeeltelijk) uit te voeren op een grafische kaart. Hierdoor vervaagt het onderscheid tussen de twee klassen contourextractie-algoritmes als het op hardware-ondersteuning aankomt. Hierop wordt vooral ingegaan in hoofdstuk 5. Desondanks blijft het efficiënt kunnen berekenen van contouren en suggestieve contouren zonder hardware-ondersteuning van groot belang: mobiele apparaten (denk aan de moderne smartphones) hebben niet altijd een geschikte grafische pipeline aan boord.

### 3.2.4 Level-Of-Detail

Bij Object Space algoritmes heeft het camerastandpunt weinig invloed op de hoeveelheid werk die moet geleverd worden om tot de uiteindelijke rendering van contourlijnen te komen. Meestal wordt voor elke vertex curvatuurinformatie berekend en worden hierop vervolgens testen uitgevoerd om te zien of er al dan niet een contour aanwezig is in de aangrenzende vlakken.

Dit is niet zo bij Image Space algoritmes: indien het camerastandpunt bijvoorbeeld relatief ver van het object verwijderd is kan met een eenvoudige pre-pass filter op de



Figuur 3.3: *View-dependent Level of Detail* bij Image Space algoritmes. Bij Object Space algoritmes zou het extraheren van contourlijnen voor beide figuren evenveel rekentijd kosten. Image Space algoritmes kunnen het scalaire veld waarop ze werken eerst verkleinen door middel van een simpele box filter die de lege cellen markeert.

render waarop signaalanalyse moet worden toegepast (de tweede stap in figuur 3.1) een deel 'lege' cellen worden genegeerd.

Dit kan bijvoorbeeld met een simpele *box filter*, die alle cellen die enkel pixels van een vooraf gekozen achtergrondkleur (in dit voorbeeld: wit) bevatten markeert. Merk hier op hoe in een Image Space algoritme de renderstijl van het scalair veld waarop gewerkt wordt kan aangepast worden aan de noden van de toegepaste signaalverwerkingsmethode. In figuur 3.3 zal de hoeveelheid werk om de rechter afbeelding te voorzien van contouren dus veel kleiner zijn dan voor de linkse afbeelding in een Image Space algoritme, terwijl de hoeveelheid werk in een Object Space algoritme dezelfde zou blijven. Deze eigenschap noemen we impliciet *view-dependent level of detail* [Rusinkiewicz, 2008]. De wisselwerking tussen de renderstijl voor het scalaire veld en de signaalverwerkingsmethode zal in hoofdstuk 5 een belangrijke rol spelen.

### 3.3 Hybride algoritmes

Voor de volledigheid wordt ook het bestaan van zogenaamde *hybride* algoritmes vermeld: deze combineren (meestal in meerdere render passes) algoritmes uit Object Space en Image Space. Zo zou in een applicatie die contourlijnen tekent de extractie van de normale contourlijnen kunnen gebeuren in Image Space en de extractie van suggestieve contourlijnen in Object Space, waarna de resultaten in een extra pass worden samengevoegd tot het finale resultaat.

Vaak maken deze algoritmes gebruik van specifieke hardware-functionaliteit en zijn ze bijgevolg veel minder algemeen toepasbaar [Raskar, 1999]. Omwille van deze redenen zullen ze in deze masterproef niet explicet aan bod komen.

### 3.4 Besluit

In dit hoofdstuk werd het kader geschetst waarin de verschillende methodes om contourlijnen te renderen die in deze thesis gebruikt worden te kunnen classificeren. Object Space algoritmes werken op de originele mesh en behouden de volledige set met contourlijnsegmenten, inclusief hun coördinaten in het assenstelsel van de originele mesh. Een gevolg hiervan is de mogelijkheid tot controle van de individuele lijndiktes in het uiteindelijke resultaat. Image Space algoritmes werken op een bepaalde afbeelding van de mesh en hebben als grote nadeel het verlies van deze controle, Daartegenover staan echter een (algemeen) verlaagde complexiteit, impliciete *level of detail*-scaling en betere hardware-ondersteuning.

## Hoofdstuk 4

# Object Space Algoritmes: CPU

In dit hoofdstuk worden verschillende algoritmes om contourlijnen en suggestieve contourlijnen te berekenen voorgesteld en vergeleken, o.a. op het gebied van correctheid en performantie. Al deze algoritmes hebben met elkaar gemeen dat alle berekeningen gebeuren op de CPU. Hiermee wordt gesteld dat er geen gebruik wordt gemaakt van specifieke GPU-functionaliteit. Ook hebben de algoritmes gemeenschappelijk dat er voor de berekeningen gewerkt wordt in Object Space, zoals gedefinieerd in sectie 3.1.

### 4.1 Technologie en begrippen

In deze sectie wordt kort besproken welke software werd gebruikt voor de implementatie van de algoritmes uit dit hoofdstuk en waarom.

#### 4.1.1 C++ en OpenGL

Als programmeertaal werd gekozen voor **C++**, een *middle-level* programmeertaal die toestaat om snel te kunnen ontwikkelen met *high-level* concepten zoals klassen en overerving, terwijl de *low-level* controle bewaard blijft waar nodig [Koenig and Moo, 2000]. Dit laatste was belangrijk omdat deze thesis zich focust op *real-time* rendering performantie, waar vaak controle over de precieze werking van basisoperaties vereist is om optimalisaties uit te kunnen voeren. De implementaties werden ontwikkeld voor Linux-x86 systemen, maar mits gebruik van een andere windowing-toolkit zijn deze gemakkelijk te porten naar andere platformen. Als compiler werd gebruikt gemaakt van de standaard GNU C++ Compiler versie 4.4.3.

**OpenGL** (*Open Graphics Library*) is een open cross-platform API voor high performance graphics [Khronos, 2006]. In de academische wereld worden de meeste (niet GPU-specifieke) referentie-implementaties aangaande real-time rendering gemaakt met behulp van de combinatie C++ en OpenGL. Als alternatief voor deze API is er Direct3D, een closed-source API voor Microsoft Windows-systemen. Om platformafhankelijkheid te maximaliseren en conform te blijven met de academische standaard werd voor OpenGL gekozen [Rosen, 2010].

De standaard OpenGL-functionaliteit is door de jaren heen uitgebreid met verschillende *extensions*, vaak ontwikkeld door hardwarefabrikanten: sommige zijn goedgekeurd door het OpenGL Architectural Review Board (ARB), maar werden nog niet opgenomen in de standaard. Alle extensies die in deze masterproef gebruikt worden verkregen goedkeuring van de ARB en worden bijgevolg ondersteund op nagenoeg alle moderne grafische hardware.

#### 4.1.2 Libraries

Aanvullend op de standaard bibliotheken en de Standard Template Library [HP, 1994] werden enkele aanvullende C++ bibliotheken gebruikt tijdens de ontwikkeling van de implementaties voor deze masterproef.

- **Trimesh2:** Een veelgebruikte library om 3D-data te manipuleren. Deze library wordt gebruikt om meshes in een datastructuur te laden en uit deze datastructuur vertex-informatie op te vragen. De viewer-implementatie van de in het kader van deze masterproef ontwikkelde tools werd gebaseerd op de tool *MeshViewer* uit hetzelfde pakket. [Rusinkiewicz, 2006]
- **FreeGLUT en GLUI:** een library die een basis windowing-toolkit en bijhorende UI-controls aanbiedt, zodat resultaten snel kunnen weergegeven worden, alsook de render parameters kunnen aangepast worden met onmiddellijk resultaat.
- **GLEW:** de *OpenGL Extension Wrangler Library*, die bij runtime kan bepalen welke OpenGL-extensies ondersteund worden door het huidige systeem in combinatie met de grafische kaart. Dit verhoogt de portability van de code, en geeft de gebruiker duidelijke informatie indien zijn systeem niet voldoet aan de vereisten voor een correcte 3D-weergave.
- **OpenMP:** API specificatie voor het gebruiken van parallelisatie in diverse programmeertalen: deze werd gebruikt om bepaalde precomputatiestappen te paralleliseren, zodat gebruik kon gemaakt worden van multi-core functionaliteit.

De code van al deze libraries (alsook die van de meegeleverde implementaties) valt onder de GPL Licentie.

#### 4.1.3 Gebruikte 3D-modellen

Om de resultaten van de verschillende implementaties te presenteren wordt gebruik gemaakt van enkele vrij verkrijgbare 3D-modellen. In tabel 4.1 worden de meestgebruikte modellen opgesomd met bronvermelding en volgens oplopend aantal polygonen. Andere gebruikte modellen werden van onbekende bron verondersteld.

Deze modellen zijn standaard testmodellen voor graphics-gerelateerde werken. Door het gebruiken van deze standaardmodellen is het eenvoudiger om de resultaten uit dit werk te vergelijken met resultaten uit andere academische literatuur.

Model	Oorsprong	Polycount
<b>Venus</b>	Mira Imaging	5 672
<b>Cow</b>	Viewpoint Animation Engineering / Sun Microsystems	92 864
<b>Horse</b>	Georgia Tech Large Geometric Models Archive	96 964
<b>Elephant</b>	Espona	157 160
<b>Armadillo</b>	Stanford 3D Scanning Repository	345 944
<b>Lion</b>	Stanford Graphics Lab	367 277
<b>Lucy</b>	Stanford 3D Scanning Repository	525 814
<b>Heptoroid</b>	UC Berkeley Rapid Prototyping Project	573 440

Tabel 4.1: Gebruikte 3D-modellen

#### 4.1.4 Benchmarks / Testmethode

Om de performantie van een bepaalde rendertechniek te bepalen, wordt (onder meer) de *framerate* van de rendering bepaald. Dit is het aantal volledige beelden die de implementatie per seconde op het scherm kan tekenen. De complexiteit van de mesh is echter niet overal dezelfde voor de 3D-modellen waarmee getest wordt: bij sommige modellen zit er bijvoorbeeld beduidend meer detail verwerkt in het hoofd dan in het lichaam van het voorgestelde dier, wat de framerate kan beïnvloeden.

Om dit effect te minimaliseren wordt het object telkens in 10.0 seconden volledig rond achtereenvolgens de x-as, y-as en z-as van het globale assenstelsel gedraaid. Elke seconde wordt het aantal frames dat succesvol werd gerenderd geteld en uitgeschreven naar een bestand. Zo krijgt men een goed gemiddelde van de prestaties over het gehele model. Tijdens deze benchmarks gelden de volgende instellingen:

- **Resolutie:** 512 bij 512 pixels.
- **Camerastandpunt:** Om de camera te positioneren in de benchmark wordt gebruik gemaakt van de *bounding box* van het 3D-model. Dit is de kleinste mogelijke balk die alle vertices van het object omvat. Het camerastandpunt wordt ingesteld zodat een vlak van de *bounding box* van het gebruikte model het volledige viewvenster vult.  
Aangezien Object Space methodes geen relevante winst halen uit het veranderen van viewpoint (zie sectie 3.2.4) wordt het camerastandpunt constant gehouden.
- **Systeem:** De tests werden uitgevoerd op een machine met een AMD Phenom II X4 processor, Radeon 4890HD grafische kaart, 2 Gb RAM en een gevirtualiseerde Ubuntu 9.10-installatie met OpenGL 2.0 en X.org versie 1.6.4

Deze methode wordt voor elke model 10 keer herhaald om tot de uiteindelijke benchmarkresultaten te komen. Op het einde van de test worden de minimum, maximum en gemiddelde framerates gerapporteerd.

## 4.2 Contouren

In deze sectie worden algoritmes behandeld die als doel hebben om normale contourlijnen, zoals gedefinieerd in sectie 2.2, te berekenen en te renderen, eventueel als overlay op een object dat met een bepaalde shading gerenderd wordt.

### 4.2.1 Algoritme: Contouren als nulpunten van $\mathbf{n} \cdot \mathbf{v}$

De formule voor de Contour Generator (2.1) geeft de voorwaarden waaraan een punt moet voldoen om op een normale contour te liggen, bekeken vanuit een bepaald camerastandpunt gedefinieerd door de vector  $\mathbf{v}$ .

Een eerste aanpak om normale contourlijnen te tekenen bestaat er dus in om de nulpunten te zoeken van (2.1) over de mesh. We veronderstellen dat deze polygonale mesh is opgebouwd uit driehoeken. Telkens wanneer het camerastandpunt wordt aangepast (en de view vector  $\mathbf{v}$  dus wijzigt) zal het volgende algoritme moeten uitgevoerd worden om de nieuwe contourlijnen te tekenen:

Men rekent eerst voor alle vertices het product  $\mathbf{n} \cdot \mathbf{v}$  uit. Vervolgens worden alle driehoeken van de mesh afgezocht: wanneer men in een driehoek een vertex heeft met een ander teken dan de twee andere vertices bevat deze driehoek een tekenwissel van de functie  $\mathbf{n} \cdot \mathbf{v}$  en bijgevolg een contourlijnsegment. In de zijdes naar de twee andere vertices wordt lineair een nulpunt bepaald. Deze twee nulpunten worden vervolgens door een rechte lijn met elkaar verbonden, om zo een segment van de contourlijn te creëren dat dwars over de behandelde driehoek loopt. Dit algoritme wordt in Pseudo Code weergegeven in codeblok 4.1.

Merk op dat de rechte lijn tussen de contourpunten een benadering vormt van het lokale contoursegment. Men kan de contourlijn verfijnen door meer nulpunten te zoeken van  $\mathbf{n} \cdot \mathbf{v}$  in het vlak, en tussen deze nulpunten een hogere-orde interpolatie uit te voeren. Eenvoudiger is echter om via een *subdividing* algoritme de mesh in meer polygonen te splitsen, zodat de contourlijnsegmenten kleiner worden. Dit heeft uiteraard wel een verhoogde rekentijd per frame als gevolg.

### 4.2.2 Alternatief algoritme: Contouren op *face edges*

Zoals [Appel, 1967] al aangaf kan voor modellen die zijn onderverdeeld in polygonen een alternatieve werkwijze toegepast worden. In plaats van per vertex de visibiliteit te bepalen, werkt men per *vlak*. Telkens wanneer het camerastandpunt wordt aangepast zal het volgende algoritme moeten uitgevoerd worden om de nieuwe contourlijnen te tekenen:

Voor alle vlakken van de mesh test men of het vlak zichtbaar is vanuit het huidige camerastandpunt. Indien het vlak zichtbaar is, worden vervolgens de aangrenzende vlakken onderzocht: indien een bepaald buurvlak niet zichtbaar is, wordt de scheidingslijn tussen dat vlak en het huidige vlak als segment van de gezochte contourlijn beschouwd. Dit algoritme wordt in Pseudo Code weergegeven in codeblok 4.2. Het algoritme vereist dat er een datastructuur is waarmee de buurvlakken eenvoudig kunnen opgezocht worden. Deze datastructuur kan via eender welke methode één-

Codeblok 4.1: NdotV Algoritme (Pseudo Code)

```

// bereken ndotv voor elke vertex
compute_vertexinfo(mesh);
// voor elk vlak van de mesh
for(face f: mesh->faces)
{
    // heeft er een vertex een verschillend teken?
    point diff, n1, n2;
    if(find_different_ndotv_sign(f, diff, n1, n2))
    {
        // in diff zit nu de vertex met het verschillend teken
        // n1 en n2 zijn de andere vertices
        float weight1 = linear_interp(mesh->ndotv[diff], mesh->ndtov[n1]);
        float weight2 = linear_interp(mesh->ndotv[diff], mesh->ndtov[n2]);
        // nulpunt bepalen (lineaire interpolatie)
        point p1 = weight1 * diff + (1.0 - weight1) * n1;
        point p2 = weight2 * diff + (1.0 - weight2) * n2;
        // segment toevoegen aan de contourlijn
        addSegment(p1,p2);
    }
}

```

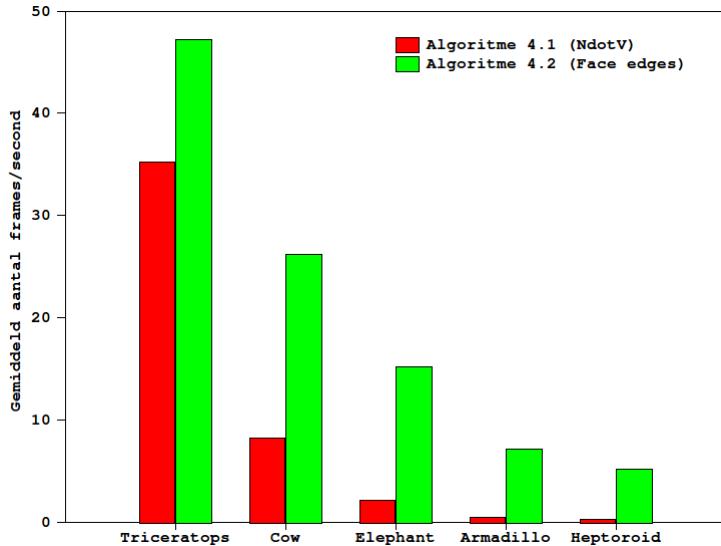
Codeblok 4.2: Face Edges Algoritme (Pseudo Code)

```

// voor elk vlak van de mesh
for(face f: mesh->faces)
{
    // is het vlak zichtbaar?
    if(isVisible(f))
    {
        // heeft dit vlak buren (niet altijd zo bij noisy meshes)
        if(hasNeighbours(f))
        {
            // zijn de buren zichtbaar?
            // Indien nee: segment toevoegen aan contourlijn
            if(!isVisible(f.neighbour0)){
                addSegment(f.vertex0, f.vertex1);}
            if(!isVisible(i.neighbour1)){
                addSegment(f.vertex1, f.vertex2);}
            if(!isVisible(i.neighbour2)){
                addSegment(f.vertex2, f.vertex0);}
        }
    }
}

```

malig opgebouwd worden in een pre-process stap, gezien deze informatie niet wijzigt gedurende het renderen.



Figuur 4.1: Grafiek met performantievergelijking van algoritmes 4.1 en 4.2. Voor complexe 3D-modellen zoals het Armadillo-testmodel en het Heptoroid-testmodel slaagt algoritme 4.1 er niet in om aan interactieve snelheid contouren te berekenen en te tekenen.

#### 4.2.3 Resultaten en evaluatie

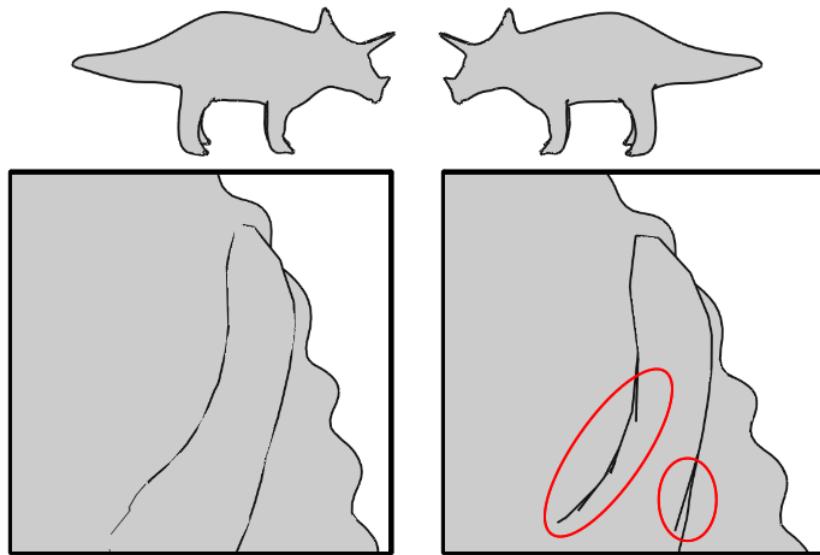
##### Performantie

Algoritme 4.2 blijkt performanter dan algoritme 4.1, zoals te zien is in grafiek 4.1. De benchmarks werden uitgevoerd voor modellen van oplopende polygon count, terug te vinden in tabel 4.1. We kunnen dit performantieverschil als volgt verklaren:

De implementatie van algoritme 4.1 is eerder triviaal, maar om te beslissen of een driehoek een contoursegment bevat en dat vervolgens te lokaliseren zijn veel (computationeel dure) if-tests nodig. Deze *bottleneck* is vooral waarneembaar bij modellen waarbij de contourlijn in veel segmenten is onderverdeeld, zijnde de meshes met veel polygonen. De performantie neemt **exponentieel** af met een oplopende polygon count van het gebruikte model. Het evalueren van  $\mathbf{n} \cdot \mathbf{v}$  op alle vertices voor elke frame is hiervan niet de oorzaak: hiervoor neemt de rekentijd slechts **lineair** toe. Deze precomputatiestap kan bovendien in parallel worden uitgevoerd voor elke vertex.

##### Correctheid

De performantiewinst bij het tekenen van contourlijnen met algoritme 4.2 komt echter niet zonder een prijs. Indien men de resultaten van beide algoritmes op macroniveau vergelijkt zijn de contouren een evenbeeld: zie bovenaan figuur 4.2. Indien men echter een detail (hier: de linkerhoorn van de Triceratops-model) rendert worden de gevonden duidelijk van de vereenvoudiging die algoritme 4.2 maakt. Door de contoursegmenten



Figuur 4.2: Detail: vergelijking resultaten algoritme 4.1 (**Links**) en 4.2 (**Rechts**) op de linkerhoorn van het Triceratops-model. Het is te zien hoe bij gebruik van het face edges-algoritme 4.2 het tekenen van contourlijnsegmenten op de scheidingslijnen tussen de verschillende polygonen zorgt voor een gebroken contourlijn.

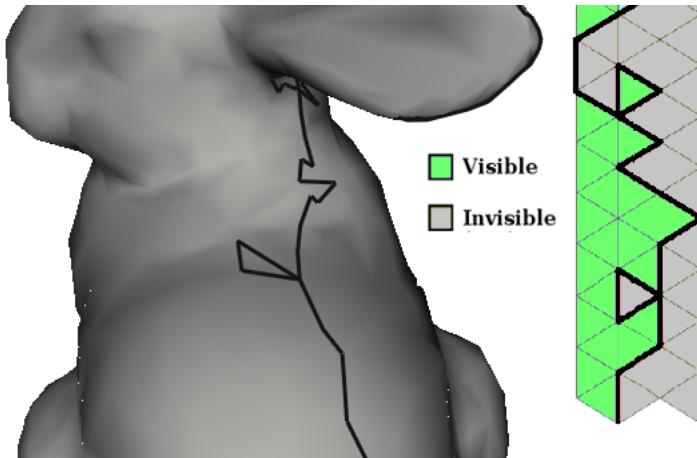
op de scheidingslijn tussen de polygonen te tekenen is bij modellen met een relatief lage polycount zichtbaar hoe deze segmenten geen sluitende benadering vormen van de contourlijn.

Bij algoritme 4.2 kan er bovendien nog een andere visuele anomalie optreden: indien de normaal  $\mathbf{n}$  van het vlak nog n t een rechte hoek maakt met de view vector  $\mathbf{v}$  wordt het vlak als 'zichtbaar' beschouwd vanuit het huidige camerastandpunt  $\mathbf{c}$ . Indien de aanliggende vlakken een hoek maken met de normaal  $\mathbf{n}$  die n t groter is dan 90 graden worden zij w l als onzichtbaar beschouwd. Dit levert 'lussen' op in de contourlijn, die vooral zichtbaar zijn bij figuren met weinig polygonen.

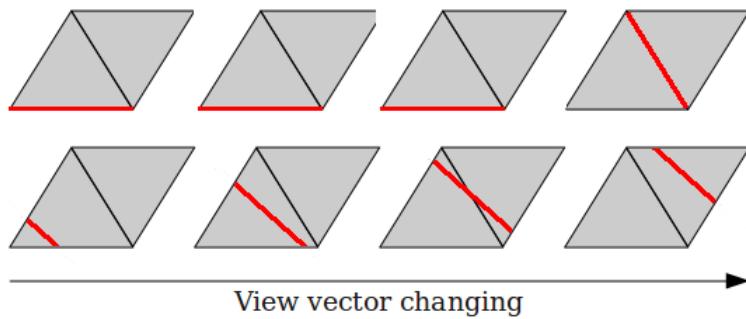
In figuur 4.3 werd ter referentie de view vector constant gehouden om de contourlijnen te kunnen weergeven vanuit een ander camerastandpunt. We zien op de rug van de Stanford Bunny (low polycount versie) een lus in de contourlijn. Omdat algoritme 4.1 contourlijnen tekent dwars over de polygonen zelf kan dit probleem daar niet voorkomen.

### Temporele coherentie

Belangrijk voor de menselijke visuele perceptie is de *temporele coherentie* van het gerenderde beeld. Dit is de mate waarin de contourlijnen stabiel blijven bij het draaien van het object, en het voor een menselijke waarnemer dus mogelijk blijft om elementen uit het beeld te beschouwen als dezelfde in de verschillende getoonde frames [Scherzer, 2010]. Merk op dat dit niets te maken heeft met het aantal frames per seconde dat de gebruiker te zien krijgt: er wordt hier enkel naar de



Figuur 4.3: Een mogelijk probleem in algoritme 4.2: er verschijnen lussen in de contourlijn indien aangrenzende vlakken een waarde voor  $n \cdot v$  hebben die zich in de buurt van het nulpunt situeert.



Figuur 4.4: Probleem met temporele coherentie van algoritme 4.2 (**Boven**) vergeleken met algoritme 4.1 (**Onder**). In het face edges algoritme 4.2 zullen in opeenvolgende frames de contourlijnen zichtbaar ‘verspringen’ over de polygonen, van scheidingslijn naar scheidingslijn.

overeenkomsten tussen frames gekeken, niet de snelheid waarmee deze gepresenteerd worden. Uiteraard helpt een hoge framerate wel om de beweging van de contourlijnen voor de menselijke hersenen als ‘vloeidend’ te interpreteren.

Bij modellen met een relatief lage polygon count zullen de contoursegmenten bij algoritme 4.2 zichtbaar ‘verspringen’ over het oppervlak, van scheidingslijn naar scheidingslijn tussen de polygonen. Omdat bij algoritme 4.1 telkens nieuwe nulpunten worden berekend om het segment op de polygoon zelf te tekenen, schuiven deze segmenten per frame geleidelijk over de polygonen in kwestie en hebben we dit storende effect dus niet. Dit verschil wordt gedemonstreerd in figuur 4.4. Ook eventuele lussen in de berekende contourlijnen, zoals weergegeven in figuur 4.3, kunnen voor storende glitches zorgen in de weergave van het model.

#### 4.2.4 Besluit

Het is duidelijk hoe algoritme 4.2 de correctheidseis verzwakt in ruil voor een betere performantie dan algoritme 4.1. De verschillende visuele problemen die gepaard gaan met het tekenen van contourlijnsegmenten op de scheidingslijnen tussen polygonen zijn echter enkel zichtbaar bij meshes met een lage polygon count. Voor interactieve toepassingen - waar snelheid meestal voorgaat op correctheid - is dus het gebruik van algoritme 4.2 aangewezen, waarbij het model eventueel eerst wordt opgedeeld in meer polygonen om deze visuele problemen te minimaliseren. Hiervoor zijn veel methodes toepasbaar, bijvoorbeeld *face subdivision* [Overveld, 1997].

### 4.3 Suggestieve contouren

In deze sectie worden algoritmes behandeld die als doel hebben om suggestieve contourlijnen zoals gedefinieerd in sectie 2.3 te berekenen en weer te geven.

#### 4.3.1 Nulpunten van $\kappa_r$ in convexe gebieden

In sectie 2.3 werden drie formules opgesteld die de Suggestieve Contour Generator beschrijven: zie secties 2.3.3, 2.3.4 en 2.3.5. Formule (2.3) geeft een goede basis voor een Object Space algoritme: eerst worden de punten gezocht waarvoor de radiale curvatuur  $\kappa_r$  gelijk is aan 0. Deze punten worden vervolgens verder gefilterd door na te gaan of het punt in een convex gebied ligt, door middel van het berekenen van de tweede directionele afgeleide in de richting van vector  $\mathbf{w}$ . Tenslotte worden de overgebleven punten gefilterd volgens de restrictie (2.8) op de hoek  $\theta$  tussen de eenheidsnormaalvector  $\mathbf{n}$  en de view vector  $\mathbf{v}$  van het punt in kwestie.

Per vertex zal men dus  $\kappa_r$  en  $D_{\mathbf{w}}\kappa_r$  efficiënt moeten kunnen berekenen. Er wordt gesteld dat volgende info over de mesh reeds berekend en beschikbaar is:

- Posities  $\mathbf{p}_i$  van alle vertices.
- Normalen  $\mathbf{n}_i$  van alle vertices.
- Principale richtingen  $\mathbf{e}_{1i}$ ,  $\mathbf{e}_{2i}$  hun curvaturen  $\kappa_{1i}$  en  $\kappa_{2i}$ , en de richtingsafgeleiden van deze twee curvaturen in beide principale richtingen,  $D_{\mathbf{e}_{1i}}\kappa_{1i}$ ,  $D_{\mathbf{e}_{2i}}\kappa_{1i}$ ,  $D_{\mathbf{e}_{1i}}\kappa_{2i}$  en  $D_{\mathbf{e}_{2i}}\kappa_{2i}$ . Deze kunnen geschat worden met het T-algoritme uit [Taubin, 1995]. Dit algoritme is een beproefde standaardmethode waarop in deze masterproef niet verder wordt ingegaan.

#### 4.3.2 $\kappa_r$ berekenen per vertex

Om  $\kappa_r$  te berekenen op elk vertexpunt  $\mathbf{p}$  kan men gebruik maken van de principale richtingen  $\mathbf{e}_1$  en  $\mathbf{e}_2$  in dat punt en hun respectievelijke curvaturen  $\kappa_1(\mathbf{p})$  en  $\kappa_2(\mathbf{p})$ . Met behulp van de formule van Euler kan dan de radiale curvatuur in  $\mathbf{p}$  als volgt berekend worden [DoCarmo, 1976]:

$$\kappa_r(\mathbf{p}) = \kappa_1(\mathbf{p}) \cos^2 \phi + \kappa_2(\mathbf{p}) \sin^2 \phi \quad (4.1)$$

waarin  $\phi$  de hoek is tussen  $\mathbf{w}$  en de normaalrichting die bij  $\kappa_1$  hoort. Alle elementen in deze formule zijn gekend voor de mesh (zie 4.3.1): formule (4.1) kan bijgevolg gebruikt worden om  $\kappa_r$  uit te rekenen.

Een andere manier om dit in te zien is door gebruik te maken van de tweede fundamentele tensor (ook wel de *Weingarten matrix* genoemd)  $\mathbf{II}$ . Men kan  $\kappa_r$  immers ook definiëren als de *normale curvatuur*  $\kappa_n$  in de richting van  $\mathbf{w}$ :

$$\kappa_r = \kappa_n(\mathbf{w}) \quad (4.2)$$

Met behulp van de de tensor  $\mathbf{II}$  kan men  $\kappa_n(\mathbf{w})$  berekenen:

$$\kappa_n(\mathbf{w}) = \frac{\mathbf{II}(\mathbf{w}, \mathbf{w})}{\mathbf{w} \cdot \mathbf{w}} \quad (4.3)$$

De notatie  $\mathbf{II}(\mathbf{w}, \mathbf{w})$  is een kortere manier om uit te drukken dat de tensor  $\mathbf{II}$  twee maal vermenigvuldigd wordt (afwisselend rechts en links) met de vector  $\mathbf{w}$ . Indien de richtingsvector  $\mathbf{w}$  uitgedrukt wordt als  $(u, v)$  in het assenstelsel bepaald door de twee principale richtingen  $\mathbf{e}_1$  en  $\mathbf{e}_2$  kan de tensor  $\mathbf{II}$  in (4.3) gediagonaalseerd worden als [Cipolla and Giblin, 2000]:

$$\mathbf{II} = \begin{bmatrix} \kappa_1 & 0 \\ 0 & \kappa_2 \end{bmatrix} \quad (4.4)$$

Uitgedrukt in het nieuwe assenstelsel is

$$\mathbf{w} = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \cos \phi \\ \sin \phi \end{bmatrix} \quad (4.5)$$

Formule (4.5) invullen in (4.3) geeft eveneens (4.1).

### 4.3.3 $D_{\mathbf{w}}\kappa_r$ berekenen per vertex

Om de richtingsafgeleide van de radiale curvatuur  $\kappa_r$  per vertex te berekenen in de richting van  $\mathbf{w}$  kan men de tweede fundamentele tensor  $\mathbf{II}$  wederom gebruiken.

In [Rusinkiewicz, 2004] wordt de 'afgeleide-van-curvatuur'-tensor  $\mathbf{C}$  gedefinieerd als een afgeleide van de tweede fundamentele tensor  $\mathbf{II}$ . Het resultaat is een  $2 \times 2 \times 2$  kubische tensor, waarvan kan bewezen worden dat die wegens de symmetrie in  $\mathbf{II}$  slechts uit 4 unieke submatrices  $P, Q, S$  en  $T$  bestaat:

$$\mathbf{C} = \begin{pmatrix} D_u \mathbf{II} & D_v \mathbf{II} \end{pmatrix} \left( \begin{pmatrix} P & Q \\ Q & S \end{pmatrix} \begin{pmatrix} Q & S \\ S & T \end{pmatrix} \right) \quad (4.6)$$

In [Rusinkiewicz, 2004] wordt eveneens een algoritme voorgesteld om de tensor  $\mathbf{C}$  te schatten per vertex met een kleinste-kwadratenbenadering van  $\mathbf{II}$ . De tweede fundamentele tensor op zich kan dan weer worden geschat uit de verschillen in normaalvectoren van de vertices van een vlak. Gezien dit een computationeel zware opdracht is (Er moet twee maal een kleinste-kwadratenbenadering uitgevoerd worden),

kan beter gebruik gemaakt worden van de informatie die voor het berekenen van  $\kappa_r$  al vereist was: de principale curvatuuren.

In [Gravesen and Ungstrup, 2002] wordt bewezen hoe we de elementen  $P, Q, S$  en  $T$  van de kubische tensor  $\mathbf{C}$  kunnen schrijven in functie van de richtingsafgeleiden van de principale curvatuuren  $\kappa_1$  en  $\kappa_2$  in beide principale richtingen  $\mathbf{e}_1$  en  $\mathbf{e}_2$ :

$$\begin{aligned} P &= D_{\mathbf{e}_1} \kappa_1 \\ Q &= D_{\mathbf{e}_2} \kappa_1 \\ S &= D_{\mathbf{e}_1} \kappa_2 \\ T &= D_{\mathbf{e}_2} \kappa_2 \end{aligned} \quad (4.7)$$

Vervolgens kan men de richtingsafgeleide in de richting van  $\mathbf{w}$  van de radiale curvatuur  $D_{\mathbf{w}} \kappa_r$  uitwerken als:

$$D_{\mathbf{w}} \kappa_r = D_{\mathbf{w}} \frac{\mathbf{II}(\mathbf{w}, \mathbf{w})}{\mathbf{w} \cdot \mathbf{w}} \quad (4.8)$$

waarbij gebruik gemaakt werd van formule (4.3). Dit is equivalent aan:

$$D_{\mathbf{w}} \frac{\mathbf{II}(\mathbf{w}, \mathbf{w})}{\mathbf{w} \cdot \mathbf{w}} = \frac{\mathbf{C}(\mathbf{w}, \mathbf{w}, \mathbf{w})}{\|\mathbf{w}\|^2} + 2\kappa_1 \kappa_2 \|\mathbf{w}\| \cot \theta \quad (4.9)$$

De extra term  $2\kappa_1 \kappa_2 \|\mathbf{w}\| \cot \theta$  verschijnt in de formule wegens het toepassen van de productregel bij het uitwerken van de directionele afgeleide  $\mathbf{II}(\mathbf{w}, \mathbf{w})$ .

In (4.9) kan men  $\mathbf{C}(\mathbf{w}, \mathbf{w}, \mathbf{w})$  uitwerken door  $\mathbf{w}$  uit te drukken als  $(u, v)$  in het assenstelsel bepaald door de twee principale richtingen  $\mathbf{e}_1$  en  $\mathbf{e}_2$ . Omdat  $\theta$  de hoek voorstelt tussen de eenheidsnormaalvector  $\mathbf{n}$  en de view vector  $\mathbf{v}$  (zie sectie 2.3.7) kan  $\cot \theta$  herschreven worden als:

$$\cot \theta = \left(1 - \left(\frac{\mathbf{n} \cdot \mathbf{v}}{\|\mathbf{v}\|}\right)^2\right)^{-1} \quad (4.10)$$

uitwerken en invullen in 4.9 geeft voor  $D_{\mathbf{w}} \kappa_r$ :

$$\frac{D_{\mathbf{e}_1} \kappa_1 u^3 + 3D_{\mathbf{e}_2} \kappa_1 u^2 v + 3D_{\mathbf{e}_1} \kappa_2 u v^2 + D_{\mathbf{e}_2} \kappa_2 v^3}{\|\mathbf{w}\|^2} + 2\kappa_1 \kappa_2 \|\mathbf{w}\| \left(1 - \left(\frac{\mathbf{n} \cdot \mathbf{v}}{\|\mathbf{v}\|}\right)^2\right)^{-1} \quad (4.11)$$

Alle elementen in deze formule zijn gekend verondersteld voor de mesh (zie 4.3.1): formule (4.11) kan bijgevolg gebruikt worden om  $D_{\mathbf{w}} \kappa_r$  uit te rekenen in een algoritme om suggestieve contouren te tekenen op een mesh. Er is geen kleinste-kwadratenbenadering of gelijkaardige curve-fitting techniek vereist om de formule toe te passen: dit probleem werd vermeden door de kubische tensor  $\mathbf{C}$  uit (4.7) te vereenvoudigen in formule (4.9).

#### 4.3.4 Algoritme: Suggestieve contouren op een mesh

Het uiteindelijke algoritme om op basis van definitie (2.3) van de Suggestieve Contour Generator de suggestieve contourlijnen te berekenen en tekenen op een mesh wordt in deze sectie gedefinieerd. Telkens wanneer het camerastandpunt wordt aangepast (en de view vector  $\mathbf{v}$  dus wijzigt) moet het volgende worden uitgevoerd om de suggestieve contourlijnen te tekenen:

Voor elke vertex worden de waarden van  $\kappa_r$  en  $D_{\mathbf{w}}\kappa_r$  met respectievelijke formules (4.1) en (4.11) berekend. Deze berekeningen kunnen in parallel gebeuren, aangezien deze onafhankelijk zijn voor elke vertex van de mesh.

Vervolgens wordt (analoog aan het algoritme 4.1 voor gewone contouren) per vlak van de mesh nagegaan of er een vertex is met een verschillend teken van  $\kappa_r$  dan de andere twee vertices. Indien dit zo is bevat het vlak in kwestie een nulpunt van  $\kappa_r$ . Vervolgens wordt de convexiteits-eis nagekeken: er wordt alleen verdergegaan voor vlakken waarvan niet alle vertices een negatieve waarde hebben voor  $D_{\mathbf{w}}\kappa_r$ .

Vervolgens worden de nulpunten van  $\kappa_r$  en  $D_{\mathbf{w}}\kappa_r$  lineair bepaald op de zijden van het vlak. Op een vlak kunnen geen, één of twee suggestieve contourlijnsegmenten liggen. Dit wordt bepaald door de ligging van de nulpunten van  $D_{\mathbf{w}}\kappa_r$  t.o.v. de nulpunten van  $\kappa_r$ . De verschillende mogelijkheden worden hieronder opgesomd en geïllustreerd aan de hand van figuur 4.5.

- Indien  $D_{\mathbf{w}}\kappa_r$  ergens in het vlak twee nulpunten heeft zullen er één of twee contourlijnsegmenten zijn: er is dan een gebied waar  $D_{\mathbf{w}}\kappa_r$  even negatief wordt, of even positief wordt. Dit zijn respectievelijk gevallen **a** en **b** in figuur 4.5
- Indien  $D_{\mathbf{w}}\kappa_r$  geen nulpunten heeft en positief is in de nulpunten van  $\kappa_r$  hebben we één contourlijnsegment. Dit is geval **c** in figuur 4.5.
- Indien  $D_{\mathbf{w}}\kappa_r$  slechts één nulpunt heeft en de nulpunten van  $\kappa_r$  liggen beiden in het negatieve gebied zijn er geen contourlijnsegmenten aanwezig. Indien één of beide nulpunten van  $\kappa_r$  in het positieve gebied liggen hebben we respectievelijk één of twee contourlijnsegmenten. Dit zijn respectievelijk gevallen **d**, **e** en **f** in figuur 4.5

Dit algoritme wordt in Pseudo Code weergegeven in codeblok 4.3. De methode `addSegment(point a, point b)` voegt een contourlijnsegment toe aan een interne lijst die alle contourlijnsegmenten bevat.

Codeblok 4.3: Suggestieve contouren op een mesh (Pseudo Code)

```

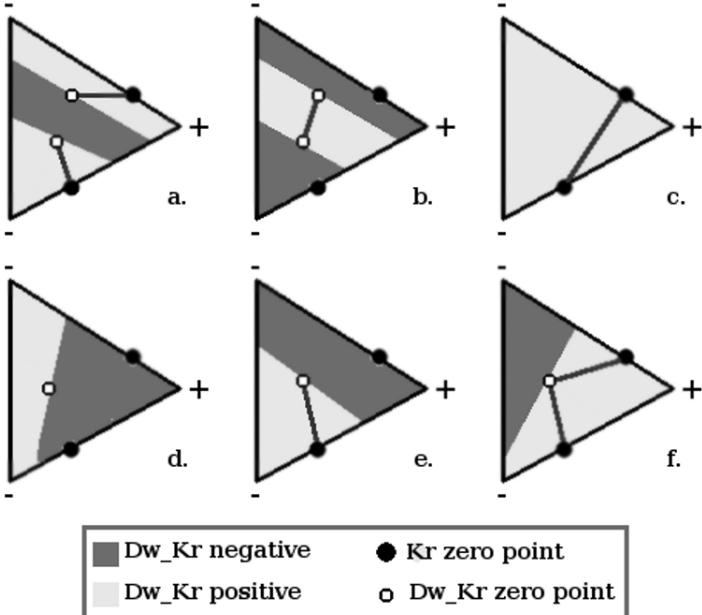
// bereken radiale curvatuur en afgeleide voor elke vertex
compute_vertexinfo(mesh);
// voor elk vlak van de mesh
for(face f: mesh->faces)
{
    // heeft er een vertex een verschillend teken van radiale curvatuur?
    point v0, v1, v2, dwkr_zero1, dwkr_zero2;
    if(find_different_kr_sign(f, v0, v1, v2))
    {
        // in v0 zit nu de vertex met het verschillend teken
        // v1 en v2 zijn de andere vertices
        float w1 = linear_interp(mesh->kr[v0], mesh->kr[v1]);
        float w2 = linear_interp(mesh->kr[v0], mesh->kr[v2]);
        // nulpunten van rad. curvatuur bepalen (lineaire interpolatie)
        point p1 = w1 * v0 + (1.0 - w1) * v1;
        point p2 = w2 * v0 + (1.0 - w2) * v2;
        // dwkr uitrekenen op de nulpunten van radiale curvatuur
        float dwkr_1 = w1 * mesh->dwkr[v0] + (1.0 - w1) * mesh->dwkr[v1];
        float dwkr_2 = w2 * mesh->dwkr[v0] + (1.0 - w2) * mesh->dwkr[v2];
        // nulpunten van dwkr zoeken
        find_zeros_dwkr(dwkr_zero1, dwkr_zero2);

        // geen nulpunten van dw_kr
        if(!dwkr_zero1 && !dwkr_zero2)
        {
            // 1 segment als dw_kr positief is in beide nulpunten van k_r
            if(dwkr_1 > 0 && dwkr_2 > 0){addSegment(p1,p2);}
        }

        // slechts 1 nulpunt van dw_kr
        else if(dwkr_zero1 && !dwkr_zero2)
        {
            // nakijken of er een nulpunt van k_r in positieve gebied ligt
            // geen, 1 of 2 segmenten
            if(dwkr_1 > 0){ addSegment(p1,dwkr_zero1);}
            if(dwkr_2 > 0){ addSegment(p2,dwkr_zero1);}
        }

        // 2 nulpunten van dw_kr
        else if(dwkr_zero1 && dwkr_zero2)
        {
            // 2 segmenten
            if(dwkr_1 > 0 && dwkr_2 > 0)
            {addSegment(p1,dwkr_zero1); addSegment(p2,dwkr_zero2);}
            // 1 segment
            else{addSegment(dwkr_zero1,dwkr_zero2);}
        }
    }
}

```



Figuur 4.5: De verschillende manieren waarop er suggestieve contourlijnsegmenten kunnen voorkomen met algoritme 4.3. Dit is afhankelijk van de nulpunten van  $D_{w\kappa_r}$ , de locatie van de nulpunten van  $\kappa_r$  en de geïnterpoleerde waarde van  $D_{w\kappa_r}$  op deze nulpunten.

#### 4.3.5 Evaluatie

##### Performantie

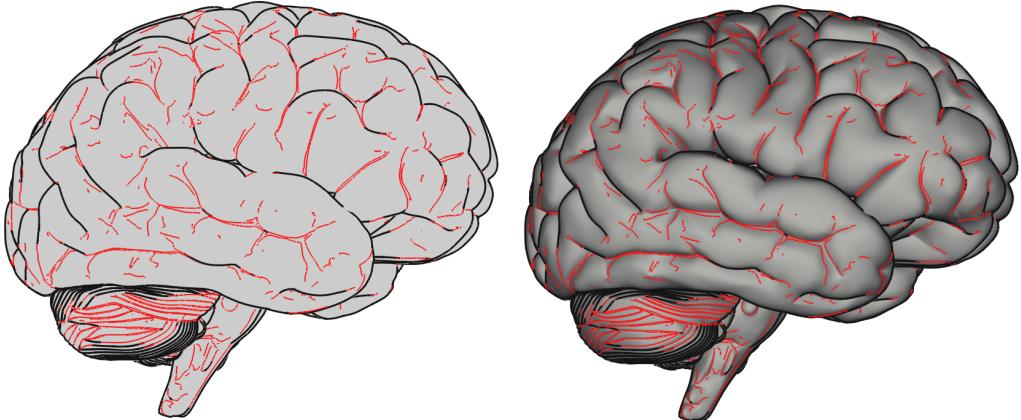
Algoritme 4.3 voor het berekenen en tekenen van suggestieve contouren op een mesh werd geïmplementeerd: de performantie werd gemeten voor het renderen van enkel suggestieve contourlijnen en voor het renderen van gewone contourlijnen samen met suggestieve contourlijnen, in twee rendering passes. Voor het berekenen van gewone contourlijnen werd algoritme 4.2 gebruikt. De resultaten van de benchmarks zijn te vinden in tabel 4.2. Een voorbeeldrendering is te vinden op figuur 4.6.

De afname in performantie die ondervonden wordt bij het tekenen van suggestieve contouren samen met gewone contouren is te verklaren door het feit dat de implementatie multi-pass is: eerst worden alle vlakken overlopen voor de gewone contouren, vervolgens nogmaals voor de suggestieve contouren. De resulterende contourlijnen worden samen getekend in de finale render.

Beide soorten contourlijnen in dezelfde pass tekenen zal in andere Object Space implementaties die gemaakt werden in het teken van deze masterproef nog aan bod komen: in dit hoofdstuk ligt de focus op het verbeteren van de afzonderlijke algoritmes 4.2 en 4.3.

Model	Polygon count	Gem. FPS (SC only)	Gem. FPS (SC+C)
Triceratops	22 460	78	43
Cow	92 864	44	19
Elephant	157 160	27	10
Armadillo	345 944	6	3
Heptoroid	573 440	4	2

Tabel 4.2: Performantie van algoritme 4.3, bij het tekenen van enkel suggestieve contourlijnen en in twee-pass combinatie met algoritme 4.2 voor het tekenen van normale contourlijnen.

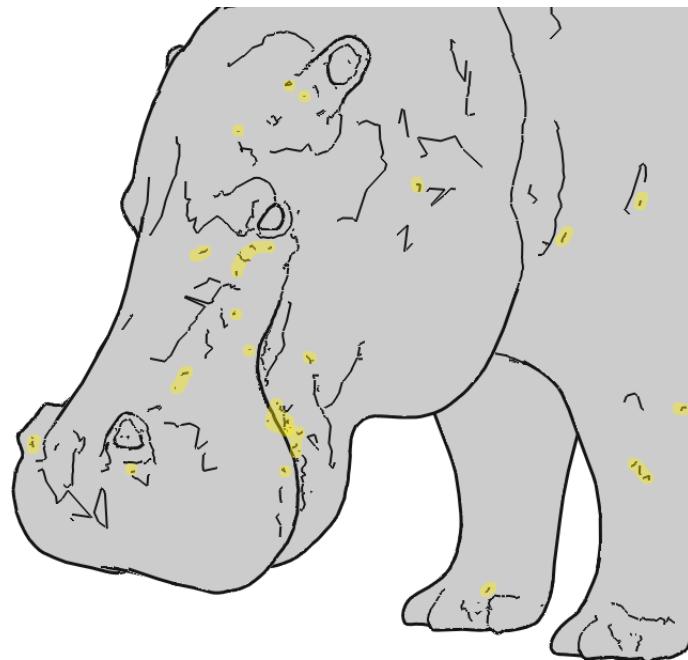


Figuur 4.6: **Links:** Gewone contouren en *Suggestieve contouren* gerenderd met algoritme 4.3. **Rechts:** Indien we een render van het model onder diffuse belichting als basis gebruiken wordt duidelijk hoe suggestieve contouren de valleien van het object volgen.

### Correctheid

Hoewel de suggestieve contouren in figuur 4.7 correct zijn volgens de definitie van de Suggestieve Contour Generator (2.3), bevinden er zich storende microcontouren op het oppervlak, aangeduid met gele markeringen. Het betreft korte, geïsoleerde suggestieve contourlijnsegmenten. Deze zijn mathematisch gezien niet ‘fout’, maar brengen weinig bij tot de perceptie van curvatuur, en kunnen dus evenwel worden weggelaten.

Deze microcontouren ontstaan door fouten in de schatting van de curvatuur op de vertexposities, waardoor  $D_{\mathbf{w}\kappa_r}$  in een klein gebied door numerieke instabiliteit positief wordt en bijgevolg een klein contoursegment getekend wordt op een enkel vlak.



*Figuur 4.7: Micro-contouren (aangeduid met gele highlights) bij toepassing van algoritme 4.3 op het Hippo-model (46 202 polygonen). Deze korte contourlijnsegmenten verschijnen kort in verschillende opeenvolgende frames en maken het moeilijker om een coherente indruk te krijgen van de curvatuur: we kunnen ze beschouwen als ruis.*

### Temporele coherentie

Indien we de temporele coherentie evalueren van de figuren gerenderd met algoritme 4.3, valt op hoe sommige suggestieve contouren ‘flickering’ gedrag vertonen: de microcontouren uit figuur 4.7 verschijnen en verdwijnen weer in de verschillende frames die worden gegenereerd bij een minimale verandering van het camerastandpunt. Dit is wederom te wijten aan de fouten in de schatting van de curvatuur op de vertexposities, waardoor ongeldige nulpunten worden gecreëerd op de vlakken van het model.

Een ander probleem vormen de suggestieve contourlijnen die snel over het oppervlak bewegen, indien de view vector  $\mathbf{v}$  snel verandert: vooral bij relatief lage framerates is het vaak moeilijk om deze contourlijnen in opeenvolgende frames te identificeren.

#### 4.3.6 Besluit

Algoritme 4.3 geeft de suggestieve contouren weer zoals die gedefinieerd werden in de eerste definitie van de Suggestieve Contour Generator (2.3.3). Er is echter ruimte voor verbetering, zowel op vlak van performantie als op vlak van temporele coherentie. Dit zal het onderwerp worden van secties 4.4 tot en met 4.6.

## 4.4 Performantieverbeteringen

Voor interactieve toepassingen (*real-time* toepassingen) is een hoge framerate belangrijk: we willen snel contourlijnen kunnen berekenen tekenen op meshes, zonder daarbij al te veel te moeten inboeten op correctheid. De toepassingen zijn talrijk en uiteenlopend van aard: computer games [Ubisoft, 2003], artistieke tools [Kalinis et al., 2002], medische applicaties [Qiang et al., 2007] ...

In de volgende secties zullen enkele verbeteringen worden voorgesteld voor algoritmes 4.2 en 4.3 op het vlak van performantie. Het basisprincipe van deze verbeteringen is altijd hetzelfde: om een significante performantiewinst te behalen, moet men ervoor zorgen dat men per frame **minder tests moet uitvoeren op minder vertices, edges en/of vlakken** om de contourlijnen te kunnen berekenen. Merk op dat we nog steeds *alle* contourlijnen wensen te vinden.

### 4.4.1 Vlakken met positieve Gaussiaanse curvatuur

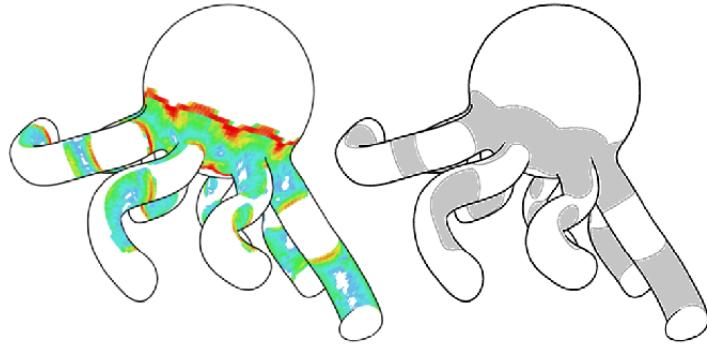
In formule 4.1 wordt gebruik gemaakt van de principale curvatuuren  $\kappa_1$  en  $\kappa_2$  om de radiale curvatuur  $\kappa_r$  te berekenen.  $\kappa_r$  kan niet negatief worden indien het product  $K = \kappa_1\kappa_2$  (de Gaussiaanse curvatuur) positief is. Indien we dus een vlak hebben waarvan de drie vertices een positieve Gaussiaanse curvatuur hebben, kan dat vlak geen suggestieve contourlijnen bevatten, omdat in dat vlak  $\kappa_r$  geen nulpunt kan bereiken.

Aangezien de principale curvatuuren onafhankelijk zijn van het camerastandpunt, kunnen deze vlakken in een eenmalige precomputatiestap al weggefilterd worden. In figuur 4.8 worden de gebieden met positieve Gaussiaanse curvatuur aangeduid op een model. In tabel 4.3 wordt bovenaan vermeld hoeveel vlakken we voor de testfiguren overhouden na het wegfilteren van de vlakken met  $K > 0$ . Merk op dat deze filtering *niet* geldt voor normale contourlijnen: die kunnen wel voorkomen op vlakken met een positieve Gaussiaanse curvatuur  $K$ .

### 4.4.2 Onzichtbare suggestieve contouren

Een andere manier om minder testen uit te moeten voeren is om suggestieve contouren die onzichtbaar zijn - omdat ze op vlakken liggen die zijn weggedraaid van het camerastandpunt - niet te tekenen. Dit proces wordt *backface culling* genoemd. We passen dezelfde test toe als in algoritme 4.2, door het dot product te berekenen tussen de normaal  $\mathbf{n}$  loodrecht op het vlak en de view vector  $\mathbf{v}$ . In tabel 4.3 worden de gemiddelde percentages van vlakken met zichtbare suggestieve contouren voor de testmodellen weergegeven.

Intuïtief zou men misschien verwachten dat er gemiddeld evenveel suggestieve contouren zichtbaar als onzichtbaar zijn vanuit verschillende camerastandpunten, maar uit de laatste lijn van tabel 4.3 blijkt dat voor onze testobjecten gemiddeld steeds minder dan de helft van de suggestieve contouren zichtbaar zijn. Het Heptoroid-model (zie laatste kolom) is hier een extreem voorbeeld van: slechts 0.5 % van zijn suggestieve contouren zijn gemiddeld zichtbaar. Een model waarbij deze ontdekking



*Figuur 4.8: Suggestieve contouren kunnen niet voorkomen in gebieden met een positieve Gaussiaanse curvatuur  $K$ . **Links:** Dichtheidsfunctie van de suggestieve contouren over alle camerastandpunten: suggestieve contouren komen het vaakst voor in gebieden waar de Gaussiaanse curvatuur  $K$  bijna 0 is (de rode gebieden). **Rechts:** De zones in het wit zijn gebieden met positieve Gaussiaanse curvatuur  $K$ . (Figuur uit [Decarlo et al., 2004])*

	Triceratops	Cow	Elephant	Armadillo	Heptoroid
Vlakken met $K < 0$	68%	65%	60%	71%	83%
Gemiddeld % vlakken met S.C.	10%	8%	16%	25%	14%
Gemiddeld % vlakken met zichtbare S.C.	4%	2%	5%	7%	0.5%
Zichtbare S.C. t.o.v totaal aantal S.C.	40%	25%	31%	28%	3%

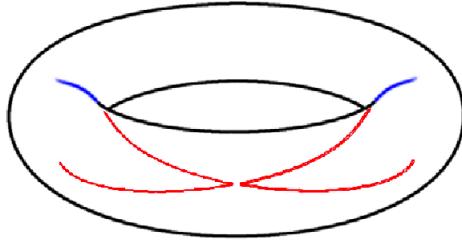
*Tabel 4.3: Filters uit secties 4.4.1 en 4.4.2 toegepast op benchmark-objecten. Het is opmerkelijk hoe door toepassing van deze eenvoudige limieten al veel vlakken kunnen worden uitgesloten in de zoektocht naar contourlijnen.*

eenvoudiger te bemerken valt is de torus (zie figuur 4.9): in [Decarlo et al., 2004] wordt bewezen hoe de onzichtbare suggestieve contouren van een torus 10 keer zo lang zijn als de zichtbare suggestieve contouren.

Een formele definitie van de klasse van geometrische figuren waarvoor dit fenomeen zich voordoet is nog niet gevonden, maar er kan wel besloten worden dat *backface culling* een aangewezen methode is om minder werk te verrichten per vlak in algoritme 4.3.

### Backface culling: plaatsing

Bij het implementeren van backface culling in algoritme 4.3 stelt zich de vraag of de backface culling test voor of na de test op de radiale curvatuur  $\kappa_r$  moet



Figuur 4.9: Bij de torus is het percentage onzichtbare suggestieve contouren (rood) veel groter dan het percentage zichtbare suggestieve contouren (blauw).

geplaatst worden. Om te testen of een vlak zichtbaar is moet een extra dot product uitgerekend worden, terwijl de radiale curvatuur  $\kappa_r$  al in een (geparalleliseerde) precomputatiestap berekend werd. Bovendien wilt men de test die het meeste aantal vlakken ‘afkeurt’ als eerste plaatsen. Op de benchmark-objecten werd getest hoeveel vlakken er gemiddeld werden afgekeurd met elke test: de resultaten zijn terug te vinden in tabel 4.4.

	Triceratops	Cow	Elephant	Armadillo	Heptoroid
$\kappa_r$ test	49%	53%	45%	42%	78%
backface culling	35%	41%	32%	37%	41%

Tabel 4.4: Gemiddeld percentage vlakken ‘afgekeurd’ door  $\kappa_r$ -test en backface culling in algoritme 4.3.

Uit tabel 4.4 is duidelijk hoe de test op  $\kappa_r$  gemiddeld meer vlakken ‘afkeurt’ en dus best vooraan geplaatst wordt in algoritme 4.3.

Waarom de backface culling-test niet gemiddeld 50 % van de vlakken wegfiltert (zoals men misschien intuïef zou verwachten) is te verklaren door de pre-filtering beschreven in sectie 4.4.1: na het wegfilteren van de vlakken met positieve Gaussiaanse curvatuur haalt men gemiddeld meer ‘winst’ door eerst vlakken weg te filteren die geen nulpunt kunnen hebben van  $\kappa_r$ . Doch is het plaatsen van deze visibiliteitstest *na* de radiale curvatuurstest niet zonder gevaren:

- Indien een mesh relatief weinig vlakken heeft met een positieve Gaussiaanse curvatuur valt het voordeel van het wisselen van de testen weg.
- In het geval waar de radiale curvaturen *on-the-fly* berekend worden (en dus niet in een geparalleliseerde precomputatiestap) is het testen op visibiliteit goedkoper en moet deze test vooraan geplaatst worden.

#### 4.4.3 Visibiliteitstesten

Met enkele (eerder eenvoudige) aanpassingen kunnen we ook de visibiliteitstesten, toegepast in algoritmes 4.2 en 4.3 versnellen:

- De normaal op een oppervlak is onafhankelijk van het camerastandpunt: deze zogenaamde *facenormals* kunnen in een precomputatiestap eenmalig berekend worden voor het model in kwestie.
- In algoritme 4.3 voor het tekenen van normale contourlijnen wordt per vlak naar de buurvlakken gekeken: door de mogelijke resultaten (zichtbaar, niet zichtbaar, contourlijn) voor deze buurvlakken met een simpele bit te markeren in een array kan ongeveer 33 % van de visibiliteitstesten uitgespaard worden.

#### 4.4.4 OpenGL optimalisaties

Door gebruik te maken van enkele features in de OpenGL-standaard kunnen OpenGL-specifieke optimalisaties ingevoerd worden - deze worden hier slechts kort besproken. Om de resultaten van de voorgaande performantieverbeteringen implementatie-onafhankelijk te houden werden deze optimalisaties ook uitgeschakeld in de uitgevoerde benchmarks.

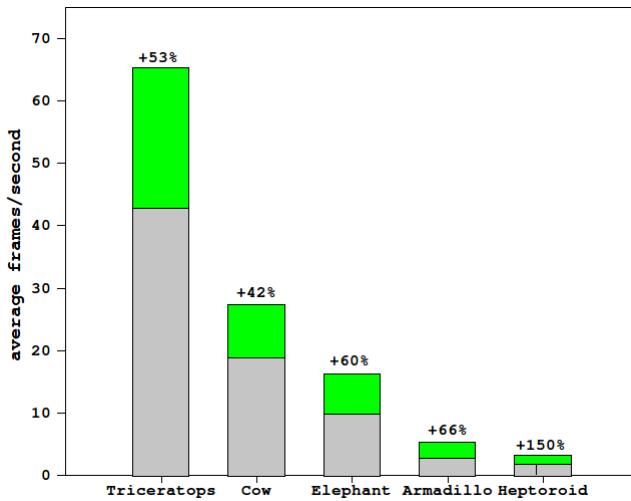
- Aangezien de vertexposities en normalen van de basis-mesh niet veranderen, kunnen deze worden opgeslagen in een Vertex Buffer Object [Ho, 2005]. Door deze data te definiëren als statische *DRAW* data, kan de hele basis-mesh éénmalig in het GPU-geheugen geladen worden, en dus in alle opvolgende frames sneller gebruikt en getekend worden.
- De door het algoritme gevonden contourlijnsegmenten kunnen in een array gebufferd worden en vervolgens met een enkele instructie getekend worden (met behulp van coordinate pointers). Zo wordt er om alle lijnsegmenten te tekenen slechts 1 API-call uitgevoerd, wat de performantie ten goede komt.

Voor meer technische details wordt naar de broncode verwezen.

#### 4.4.5 Evaluatie

Na het implementeren van de verbeteringen voorgesteld in secties 4.4.1 - 4.4.4 werden de benchmarks opnieuw uitgevoerd voor het gecombineerd tekenen van gewone en suggestieve contouren met respectievelijk algoritmes 4.2 en 4.3. De resultaten worden weergegeven in de grafiek op figuur 4.10.

Door het invoeren van de verbeteringen wordt gemiddeld een performantiewinst van 40 tot 60% gehaald. De eerder spectaculaire winst van 150% bij het *heptoroid*-testobject is te wijten aan het feit dat deze figuur gemiddeld weinig front-facing suggestieve contouren heeft, zoals eerder werd aangegeven in tabel 4.3. De output van de algoritmes is exact dezelfde gebleven: er worden simpelweg minder vlakken getest en/of de tests op een vlak worden sneller afgebroken. Nog steeds worden *alle* zichtbare contourlijnsegmenten en suggestieve contourlijnsegmenten gevonden en getekend.



Figuur 4.10: Performantieverbeteringen voor het tekenen van normale contourlijnen met algoritme 4.2 en suggestieve contourlijnen met algoritme 4.3 in een multi-pass implementatie na het toevoegen van de performantieverbeteringen uit secties 4.4.1 - 4.4.4.

Het opzet van deze sectie is bijgevolg geslaagd: er is een performantieverbetering mogelijk zonder daarbij ook maar iets te moeten inboeten aan de correctheid van het resultaat.

## 4.5 Stochastische technieken

Een tweede manier om minder vlakken te moeten bezoeken bij het berekenen van de contourlijnsegmenten en zo de efficiëntie van algoritmes 4.2 en 4.3 te verbeteren is zo fundamenteel verschillend dat ze behandeld dient te worden in een afzonderlijke sectie.

Bij zogenaamde stochastische algoritmes worden er slechts een beperkt *willekeurig* aantal vlakken getest om zo contouren te vinden. Zelden zullen met deze methodes *alle* contoursegmenten gevonden worden, maar mits enkele aanpassingen kan de meerderheid van de segmenten ontdekt worden, waardoor de visuele impact minimaal blijft.

### 4.5.1 Theorie

In [Markosian et al., 1997] wordt een stochastisch algoritme voorgesteld om te zoeken naar nulpunten van  $\mathbf{n} \cdot \mathbf{v}$  over de mesh voor contouren op de scheidingslijnen tussen polygonen, zoals in algoritme 4.3. In [Kalinis et al., 2002] wordt dit algoritme uitgebreid om te werken op de vlakken zelf, zoals in algoritme 4.1. We kunnen deze techniek ook toepassen om suggestieve contouren te vinden.

Codeblok 4.4: Markosian Stochastisch Algoritme (Pseudo Code)

```

bool visited[amount];
// kies een subset van de vlakken
choose_random_faces(faces, amount);
// voor elk gekozen vlak
for(face f: faces){
    search_contour(f);
}

search_contour(face f){
    // als we het vlak nog niet bezocht hebben
    if(!visited[f]){
        visited[f] = true;
        //en er is een contour
        if(!visited[f] && containscontour(f)){
            // teken de contour
            drawcontour(f);
            // volg de contour (via de buurvlakken)
            search_contour(f.neighbour0);
            search_contour(f.neighbour1);
            search_contour(f.neighbour2);
        }
    }
}

```

#### 4.5.2 Basisalgoritme

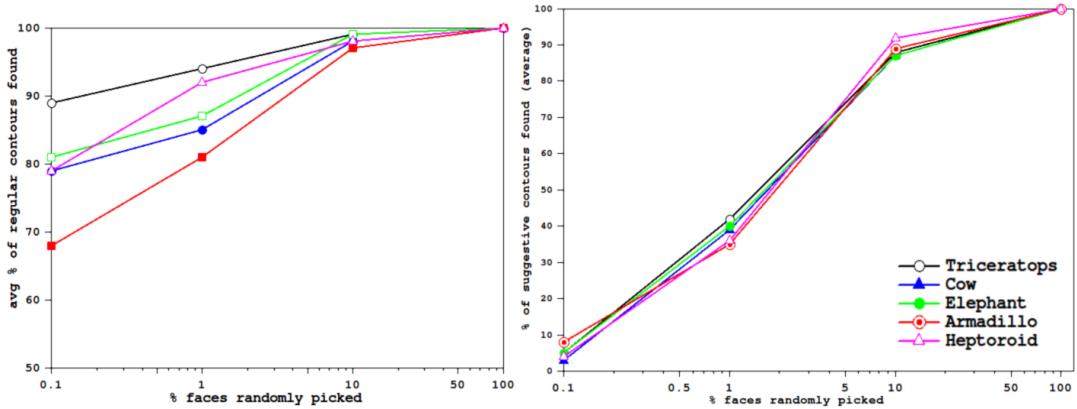
Het basisalgoritme wordt als volgt beschreven:

Er wordt een willekeurig vlak van de mesh gekozen. Dit vlak wordt getest op een nulpunt van  $\mathbf{n} \cdot \mathbf{v}$  (voor contouren) en de radiale curvatuur  $\kappa_r$  (voor suggestieve contouren). Indien in dat vlak een (suggestieve) contour gevonden werd, wordt de contour ‘gevolgd’ tot een vlak wordt tegengekomen dat al bezocht werd. Dit wordt praktisch gerealiseerd met een recursieve methode, zoals gedemonstreerd in de Pseudo Code in codeblok 4.4.

In [Markosian et al., 1997] wordt bewezen hoe voor een voldoende grote mesh de extra complexiteit van het testen van een willekeurig deel van de vlakken en het volgen van de contourlijnen kleiner wordt dan de complexiteit bij het bezoeken van *alle* vlakken.

De resultaten voor respectievelijk contouren en suggestieve contouren bij het toepassen van het basis stochastisch algoritme 4.4 worden weergegeven in figuur 4.11. Voor de percentages op de y-as werd telkens het gemiddelde percentage van gevonden contoursegmenten over de hele benchmark genomen.

Het is opmerkelijk hoe bij gewone contouren bij het gebruik van 1% van de vlakken reeds meer dan 80% van de contoursegmenten gevonden wordt, terwijl voor de suggestieve contouren bij hetzelfde percentage niet eens de helft van de suggestieve contoursegmenten gevonden worden, zoals te zien rechts in figuur 4.11.



Figuur 4.11: Benchmarkresultaten bij toepassen van basisalgoritme 4.4 voor gewone contouren (**links**) en suggestieve contouren (**rechts**), met een oplopend aantal vakken willekeurig gekozen: v.l.n.r. 0.1%, 1%, 10%, 50% en 100% van de vlakken.

Dit is te verklaren door het feit dat normale contourlussen (en dus ook de geldige segmenten daarvan) over het algemeen langer en minder talrijk zijn dan suggestieve contourlussen, die vaak slechts over een handvol vlakken een lus maken. Indien we dus een contoursegment gevonden hebben, kunnen we voor gewone contouren veel langer nieuwe segmenten ontdekken dan voor suggestieve contouren, waar we relatief snel in een reeds bezocht vlak terechtkomen en de recursie in algoritme 4.4 afbreken.

### 4.5.3 Verbeteringen

#### Gebruik van seeds

In [Markosian et al., 1997] wordt voor gewone contourlijnen een verbetering voorgesteld: door de vlakken waar in het vorige gerenderde frame een contour werd gevonden bij te houden, en in het volgende frame een subset van deze vlakken als *seeds* te gebruiken voor het vinden van contoursegmenten verhoogt men de kans op het vinden van contourlijnsegmenten: indien de segmenten aangeduid door de *seeds* onderdelen waren van een reeks contourlijnsegmenten in een vorig frame, is de kans groot dat ze opnieuw tot de ontdekking van deze reeks contourlijnsegmenten zullen leiden. Deze veronderstelling vervalt uiteraard wanneer het camerastandpunt zeer drastisch werd aangepast t.o.v. de frame waarin de *seeds* verzameld werden.

Deze redenering kan ook toegepast worden voor suggestieve contouren, waarvan de meeste (zie sectie 4.3.5 voor het probleem van ‘flickering’ microcontouren) relatief stabiel zijn, en we dus ook vlakken uit de vorige frame als *seeds* kunnen gebruiken om in de huidige frame sneller suggestieve contourlijnen te vinden.

#### Aanpassingen algoritmes

Het feit dat we willekeurige vlakken kiezen om zowel contouren als suggestieve contouren te tekenen heeft enkele praktische gevolgen voor de reeds gedefinieerde

algoritmes 4.2 en 4.3:

- De precomputatiestap voor  $\mathbf{n} \cdot \mathbf{v}$ ,  $\kappa_r$  en  $D_{\mathbf{w}}\kappa_r$  wordt weggehaald: deze waarden worden nu *on-the-fly* berekend. Indien we slechts een klein percentage van de vlakken gaan bezoeken zou het inefficiënt zijn om in een precomputatiestap voor *alle* vlakken deze informatie te berekenen.
- Aangezien contouren en suggestieve contouren samen getekend worden kunnen ze dezelfde *pool* van random geselecteerde vlakken gebruiken: voor suggestieve contouren worden de vlakken met positieve Gaussiaanse curvatuur  $K$  eerst weggefiterd (zie sectie 4.4.1). De random vlakken voor het bepalen van suggestieve contouren zullen uit deze gereduceerde subset gekozen worden. Dit verhoogt de kans aanzienlijk om een vlak te treffen waarin zich een suggestief contourlijnsegment bevindt. Voor gewone contouren moet uit de volledige set vlakken gekozen worden: zij kunnen immers wel voorkomen op vlakken met een positieve waarde voor  $K$ .

### Discovery parameter

Het gebruik van *seeds* wordt minder efficiënt als het camerastandpunt sneller verandert: in opeenvolgende frames zullen dan slechts weinig van deze *seeds* een aanleiding geven tot een reeks suggestieve contoursegmenten. Tijdens de implementatie van algoritme 4.4 en de verbeteringen uit sectie 4.5.3 ontstond dan ook de behoefte voor een parameterwaarde om te kunnen variëren hoeveel aandacht er moet besteed worden aan de set met *seeds*.

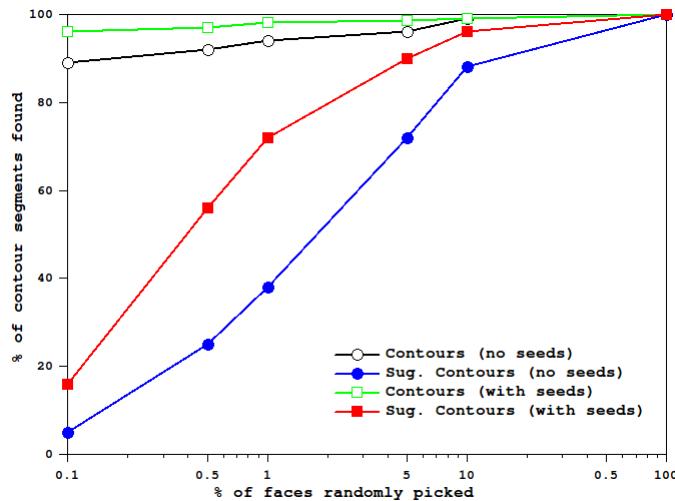
Hiervoor werd de *discovery* parameter  $\delta$  ingevoerd. Indien  $n$  het aantal gewenste vlakken voorstelt zullen er in elke frame  $(1 - \delta)n$  seeds en  $(1 + \delta)n$  random vlakken gebruikt worden. Door het variëren van de  $\delta$  parameter tussen 0 en 1 kan men dus als het ware seeds ‘inruilen’ voor een extra set willekeurige vlakken, hetgeen bij een snelle beweging van het camerastandpunt nuttiger is dan (met een hoge graad van waarschijnlijkheid ongeldig geworden) seeds testen.

Hoewel voor de relatie tussen  $\delta$  en de rotatiesnelheid  $v$  van het object geen formeel bewijs of formule gevonden werd, kunnen we proefondervindelijk wel vaststellen dat deze parameter toelaat om meer contouren te vinden, afhankelijk van de rotatiesnelheid: het Triceratops-model werd getest op normale snelheid van de benchmark, en vervolgens op vijfvoudige snelheid, dit eenmaal met  $\delta$  gelijk aan 0.0 (alle seeds worden getest) en eenmaal met  $\delta$  gelijk aan 0.8 (80% van de seeds wordt genegeerd in ruil voor meer willekeurige vlakken). De resultaten zijn te vinden in tabel 4.5.

We zien dat voor een relatief lage rotatiesnelheid de waarde  $\delta = 0.0$  in het aangepaste algoritme 4.4 een hoger percentage suggestieve contouren vindt: de seeds zijn hier nuttig omdat het camerastandpunt slechts met kleine stappen verandert en er dus telkens maar een klein aantal seeds ongeldig wordt. Vice versa is voor een hoge rotatiesnelheid de waarde  $\delta = 0.8$  beter.

	$\delta = 0.0$	$\delta = 0.8$
rotation speed $\times 1$	91%	54%
rotation speed $\times 5$	42%	81%

Tabel 4.5: Percentage van suggestieve contouren gevonden bij verschillende rotatiesnelheden van het Triceratops-model, met verschillende waarden voor de discovery parameter  $\delta$ . Bij een hoge rotatiesnelheid is het beter om meer nieuwe vlakken te onderzoeken dan seeds uit een vorige frame te testen.

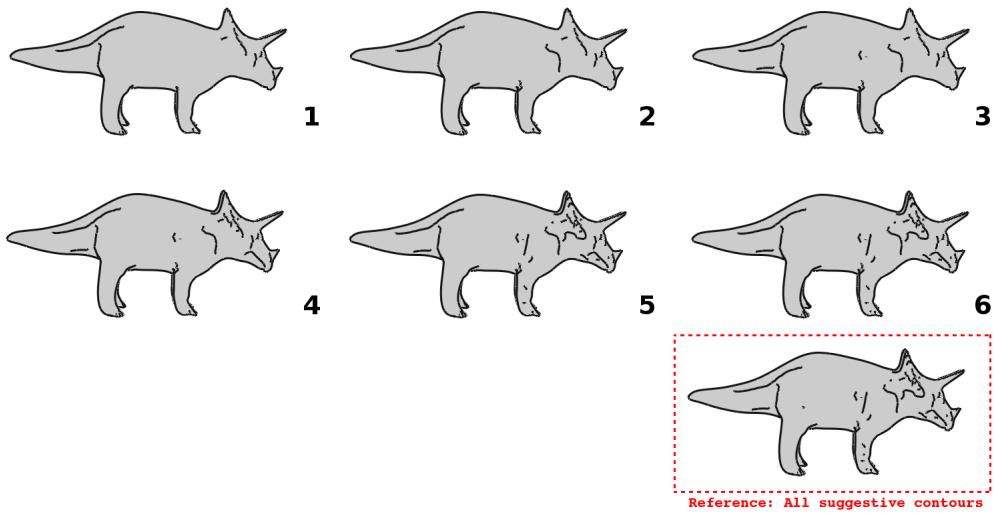


Figuur 4.12: Benchmarkresultaten bij toepassen van verbeteringen uit sectie 4.5.3 voor het Triceratops-model. Het object werd geroteerd aan de normale benchmark-snelheid, met discovery parameter  $\delta = 0.0$ . We zien dat met het gebruik van seeds veel meer contourlijnsegmenten gevonden worden bij eenzelfde aantal willekeurig geselecteerde vlakken.

#### 4.5.4 Evaluatie

Na het implementeren van de verbeteringen uit sectie 4.5.3 kan men de percentages gevonden contourlijnsegmenten vergelijken met die van het originele algoritme 4.4. De vergelijking van de resultaten voor het Triceratops-model zijn te vinden in grafiek 4.12. Het is duidelijk hoe het gebruik van seeds bij het zoeken naar suggestieve contouren zijn vruchten afwerpt, zeker indien we slechts een kleine fractie van de vlakken gebruiken. Aangezien lussen van gewone contouren over het algemeen langer zijn behalen we reeds bij zeer kleine fracties van gekozen vlakken in combinatie met seeds goede resultaten.

In verband met temporele coherentie kan opgemerkt worden dat in het geval waar een (suggestieve) contour ‘gemist’ wordt in een bepaald frame, deze waarschijnlijk gevonden zal worden in de volgende frame. Een voorbeeld voor het Triceratops-model



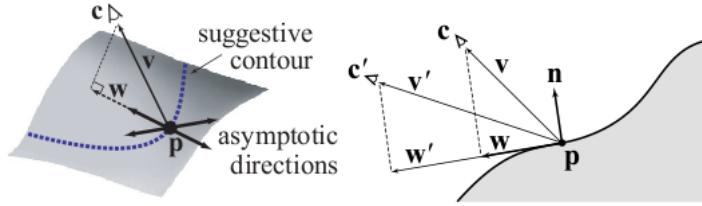
*Figuur 4.13: Temporele coherentie bij 6 opeenvolgende frames in het stochastisch algoritme 4.4. Referentie-afbeelding rechtsonder: alle contouren getekend. Er werd 1% van de beschikbare vlakken gebruikt. We zien dat na slechts 6 frames al ongeveer alle contourlijnen gevonden zijn dankzij het bijhouden van seeds uit de vorige frames.*

is te vinden in figuur 4.13. Bij het kiezen van slechts 1% van de vlakken (in dit geval: 225 vlakken van de 22460) zien we hoe in de eerste 6 frames op incrementele wijze (dankzij het bijhouden van de seeds uit de vorige frames) het overgrote deel van de suggestieve contouren gevonden wordt. Ter vergelijking werd een render opgenomen van het Triceratops-model met alle suggestieve contouren zichtbaar (rechtsonder in figuur 4.13). Na zes frames is de met het stochastisch algoritme incrementeel opgebouwde render al (op enkele details na) zeer goed vergelijkbaar met de ‘correcte’ oplossing.

Indien we dus met stochastische algoritmes contoursegmenten tekenen en dit aan een framerate kunnen die hoog genoeg ligt zullen deze tijdelijke onvolmaakthesen niet of nauwelijks worden waargenomen, en blijft de temporele coherentie binnen aanvaardbare grenzen.

#### 4.5.5 Besluit

Stochastische algoritmes vormen een waardevol alternatief voor de technieken uit secties 4.2 en 4.3. Het gebruik van *seeds* zorgt voor een grote verbetering in het aantal gevonden contoursegmenten en introduceert een aantal interessante vragen en problemen waarop verder onderzoek mogelijk is, zoals het afwegen van het gebruik van deze seeds tegenover nieuwe vlakken (in deze masterproef benaderend opgelost door het invoeren van de *discovery* parameter  $\delta$ ) en het performant opslaan en selecteren van deze seeds.



Figuur 4.14: Indien de camerapositie  $\mathbf{c}$  enkel beweegt in het radiale vlak beweegt het punt  $\mathbf{p}$  op de suggestieve contour niet: de vector  $\mathbf{w}$  verandert immers enkel van grootte, en niet van richting. (Figuur uit [Decarlo et al., 2004])

## 4.6 Temporele coherentie verbeteren

In sectie 4.2.3 werd het probleem van *temporele coherentie* kort aangehaald: men wenst om in opeenvolgende frames bij veranderende camerastandpunten bepaalde elementen, in dit geval contoursegmenten, uit de verschillende frames te beschouwen als ‘dezelfde’. In de volgende secties wordt een beschrijving voor de bewegingen van suggestieve contoursegmenten in opeenvolgende frames opgesteld en worden enkele technieken voorgesteld om de temporele coherentie tussen opeenvolgende frames te verbeteren, zonder daarbij te moeten inboeten aan performantie. De volgende secties zijn gebaseerd op [Decarlo et al., 2004], een follow-up paper op de originele basis-paper over suggestieve contouren, [Decarlo et al., 2003].

### 4.6.1 Snelheid van contoursegmenten

Suggestieve contouren zijn afhankelijk van  $\mathbf{w}$ , de op het radiaal vlak neergeslagen view vector  $\mathbf{v}$ , die gedefinieerd werd als  $\mathbf{c} - \mathbf{p}$ , waarin  $\mathbf{c}$  de locatie is van de camera en  $\mathbf{p}$  de positie van het beschouwde punt. We kunnen dus verwachten dat suggestieve contouren bewegen als het camerastandpunt wordt aangepast. Dit gebeurt enkel als  $\mathbf{w}$  van *richting* verandert: indien we de camera bewegen in het radiale vlak (zie sectie 2.3.5) verandert  $\mathbf{w}$  enkel van *lengte*, en beweegt het contoursegment dus niet. Dit wordt gedemonstreerd op figuur 4.14.

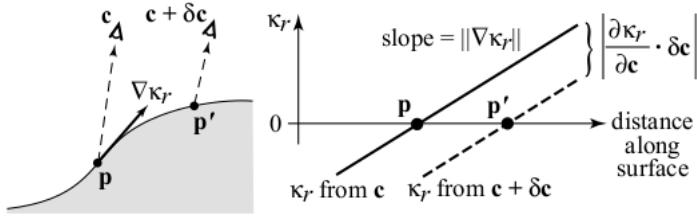
Indien een punt wel beweegt op de contour hangt zijn beweging van twee elementen af: de helling van de curve, aangeduid door de lokale gradiënt van de radiale curvatuur  $\|\nabla\kappa_r\|$ , en de invloed van de cameraverandering  $\delta\mathbf{c}$  op de radiale curvatuur, aangeduid door  $|\frac{\kappa_r}{\delta\mathbf{c}} \cdot \delta\mathbf{c}|$ . Dit wordt gedemonstreerd op figuur 4.15.

Met behulp van het impliciete functietheorema uit [Munkres, 1991] kan bewezen worden dat de snelheid van een punt op een suggestieve contour voor een camerabeweging  $\delta\mathbf{c}$  bepaald wordt door:

$$v_{sc} = -\frac{\left(\frac{\kappa_r}{\delta\mathbf{c}} \cdot \delta\mathbf{c}\right)}{\|\nabla\kappa_r\|} \quad (4.12)$$

met als maximumwaarde voor  $v_{sc}$ :

$$\max|v_{sc}| = -\frac{\left\|\frac{\kappa_r}{\delta\mathbf{c}}\right\|}{\|\nabla\kappa_r\|} \quad (4.13)$$



Figuur 4.15: Beweging van het punt  $p$  op een suggestieve contour. (Figuur uit [Decarlo et al., 2004])

Deze formule kan in theorie gebruikt worden om suggestieve contouren die té snel bewegen te filteren uit de gevonden contoursegmenten, maar in praktijk blijkt dit niet zo eenvoudig te zijn. Het evalueren van uitdrukking 4.13 voor elke vertex van de mesh is computationeel zwaar en weinig geschikt voor real-time toepassingen: om de snelheid van de eindpunten van de contoursegmenten te berekenen zijn vierde afgeleiden over een oppervlak nodig. Bovendien is voor discrete meshes de nauwkeurigheid op het vierde niveau van afleiding klein, gezien in elke afleidingsstap een benadering wordt gemaakt van de continue mesh: een kleine hoeveelheid ruis kan hier al snel roet in het eten gooien. Daarom opteren we in deze masterproef voor een andere (eenvoudigere) limiet om de temporele coherentie te verbeteren.

#### 4.6.2 Ongeldige nulpunten van $D_{\mathbf{w}}\kappa_r$

Zoals in sectie 4.3.5 werd aangehaald en gedemonstreerd op figuur 4.7 vormen zogenaamde *micro-contouren* een probleem voor de temporele coherentie in opeenvolgende frames: deze korte segmenten worden veroorzaakt door een foutieve schatting van  $D_{\mathbf{w}}\kappa_r$  op een vlak, waardoor er ongeldige nulpunten ontstaan: tussen deze nulpunten is  $D_{\mathbf{w}}\kappa_r$  positief en wordt een micro-contoursegment getekend. Door de numerieke instabiliteit van de schatting van  $D_{\mathbf{w}}\kappa_r$  kunnen deze microcontouren in opeenvolgende camerastandpunten  $\mathbf{c}$  een storend ‘flickering’ gedrag vertonen.

Aangezien deze kleine contoursegmenten weinig toedragen tot de perceptie van curvatuur van het oppervlak wenst men deze weg te filteren. Naar voorbeeld van [Hildreth, 1985] wordt in [Decarlo et al., 2003] de parameter  $t_d$  voorgesteld, een limiet die eist dat  $D_{\mathbf{w}}\kappa_r$  voldoende positief is voordat de eventueel gevonden nulpunten aanleiding geven tot een contoursegment:

$$t_d < \frac{D_{\mathbf{w}}\kappa_r}{\|\mathbf{w}\|} \quad (4.14)$$

Het is vanuit het oogpunt van een implementatie wenselijk dat deze limiet onafhankelijk is van de grootte van de gebruikte mesh. De gebruikte curvaturen  $\kappa_1$  en  $\kappa_2$  zijn echter niet schaal-invariant [Guggenheim, 1977]. Daarom worden alle curvaturen vermenigvuldigd met een zogenaamde *feature size*  $f_s$ , een waarde die als maatstaf dient voor de grootte van de mesh. In de implementatie van deze masterproef wordt gebruik gemaakt van de mediaan van de lengtes van de scheidingslijnen tussen

vertices, maar er zijn andere mogelijkheden (grootte van de bounding box, omhullende bol, ...). Omwille van de productregel bij afgeleiden wordt  $D_{\mathbf{w}}\kappa_r$  tweemaal vermenigvuldigd met de feature size  $f_s$ .

#### 4.6.3 Limieten combineren

In sectie 2.3.7 werd reeds een limiet ingevoerd op  $\theta$ , de hoek tussen de eenheidsnormaalvector  $\mathbf{n}$  en de view vector  $\mathbf{v}$  op het punt  $\mathbf{p}$ . In [Decarlo et al., 2004] wordt aangegeven hoe deze limiet kan gecombineerd worden met de limiet uit (4.14) tot limiet  $t_d$ , zonder daarbij te moeten inboeten aan controle over de filtering van contoursegmenten:

$$t_d < \sin^2 \theta \frac{D_{\mathbf{w}}\kappa_r}{\|\mathbf{w}\|} \quad (4.15)$$

waarbij de test uit sectie 2.3.7 vervat zit in  $\sin^2 \theta$ :

$$\sin^2 \theta = \frac{\mathbf{w} \cdot \mathbf{w}}{\mathbf{v} \cdot \mathbf{v}} = 1 - (\frac{\mathbf{n} \cdot \mathbf{v}}{\|\mathbf{v}\|})^2 = \cos^2(\frac{\mathbf{n} \cdot \mathbf{v}}{\|\mathbf{v}\|}) \quad (4.16)$$

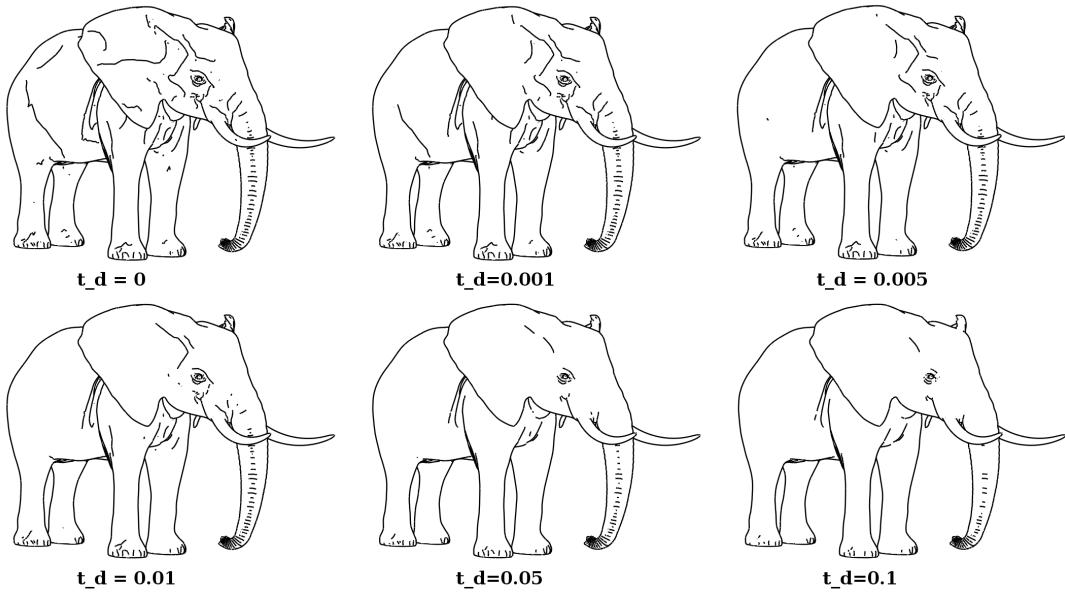
Het samenvoegen van deze limiet zorgt ervoor dat er slechts één parameter nodig is om de filtering van de contouren op de mesh te kunnen limiteren. Dat beide limieten mogen samengevoegd mogen worden kan intuïtief worden bewezen door 4.15 te vermenigvuldigen met  $(\mathbf{v} \cdot \mathbf{v}) \|\mathbf{w}\|$ :

$$\sin^2 \theta \frac{D_{\mathbf{w}}\kappa_r}{\|\mathbf{w}\|} (\mathbf{v} \cdot \mathbf{v}) \|\mathbf{w}\| = (\mathbf{w} \cdot \mathbf{w}) D_{\mathbf{w}}\kappa_r = D_{\mathbf{w}}((\mathbf{w} \cdot \mathbf{w})\kappa_r) = D_{\mathbf{w}}(D_{\mathbf{w}}(\mathbf{n} \cdot \mathbf{v})) \quad (4.17)$$

Opmerkelijk is dat (4.17) equivalent blijkt aan de tweede definitie van de Suggestieve Contour Generator (2.5), die de verzameling punten beschreef als minima van  $(\mathbf{n} \cdot \mathbf{v})$  in de richting van  $\mathbf{w}$ . Waar  $\mathbf{n} \cdot \mathbf{v}$  klein is, zal de hoek  $\theta$  klein zijn, en  $\mathbf{w}$ , de projectie van  $\mathbf{v}$  op het radiale vlak, zal een kleine lengte  $\|\mathbf{w}\|$  hebben, waardoor  $(\mathbf{w} \cdot \mathbf{w})\kappa_r$  ook klein is. Dit verklaart waarom het samenvoegen van de twee limieten geen afbreuk doet aan hun afzonderlijke kracht. De invloed van de limiet op het aantal weergegeven contourlijnen wordt voor het Olifant-model gedemonstreerd in figuur 4.16. Het is te zien hoe reeds bij een kleine waarde voor  $t_d$  (0.001) de meeste microcontouren worden verwijderd uit de figuur.

#### 4.6.4 Fading

Door het toepassen van limiet (4.15) worden een deel van de in sectie 4.3.5 aangehaalde *micro-contouren* weggefilterd, maar een ander probleem blijft bestaan: door het toepassen van een ‘harde’ limiet worden sommige contourlijnen opgedeeld in verschillende kleinere segmenten. Deze contourlijnen kunnen, afhankelijk van het camerastandpunt, in verschillende sets segmenten opgebroken worden in verschillende frames, wat de temporele coherentie niet ten goede komt. In [Decarlo et al., 2003] werd dit opgelost door een Image Space post-processing functie die kleine onderbrekingen in contourlijnen opvult - dit is in het kader van real-time oplossingen echter niet optimaal: een extra pass is vereist.



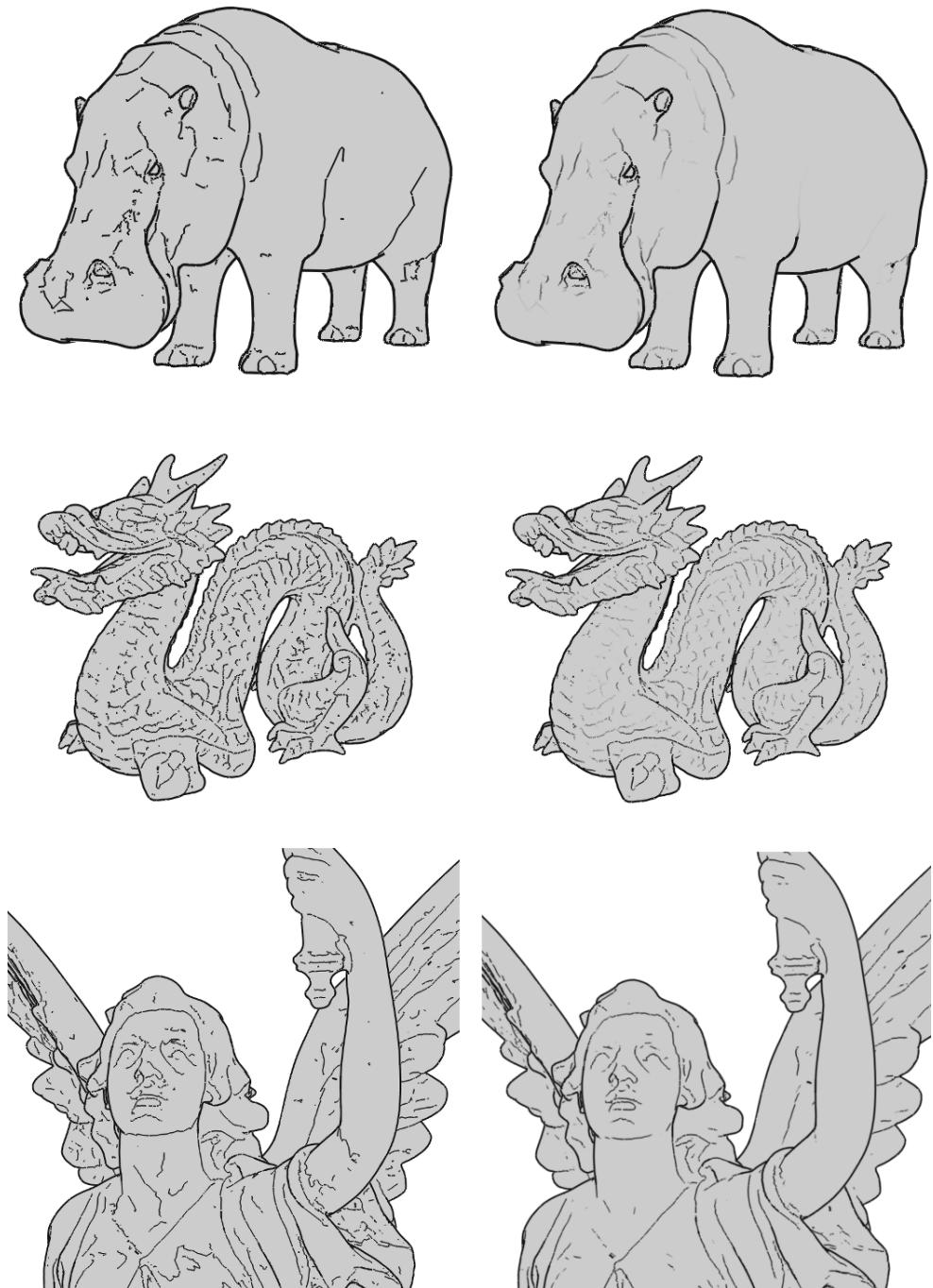
Figuur 4.16: Invloed van gecombineerde limietwaarde  $t_d$  op het aantal weergegeven contourlijnen. Linksboven is de waarde voor  $t_d = 0$ : hier wordt de ongefilterde figuur weergegeven.

Het ‘tracken’ van contourlijnen over verschillende camerastandpunten om zo de contourlijn ononderbroken te houden, zoals in [Kalnins et al., 2003] is erg complex en evenmin geschikt voor toepassingen waarin de performantie van groot belang is.

Daarom wordt voorgesteld om een simpel *fading* schema in te voeren: hoe verder de eindpunten van een contoursegment positief verwijderd zijn van de limiet (4.15), hoe donkerder ze getekend worden. Alle punten daartussen variëren lineair in kleur tussen de kleuren gedefinieerd in de eindpunten voor het fading schema. Deze simpele oplossing zorgt ervoor dat instabiele contoursegmenten niet of slechts zeer licht getekend worden, waardoor hun storende ‘flickering’ effect geminimaliseerd wordt. Bovendien is deze oplossing computationally zeer voordelig: alle informatie die vereist is voor de berekening van de fading-coefficiënt is al aanwezig in algoritme 4.3.

#### 4.6.5 OpenGL Optimalisaties

Tenslotte kunnen we door gebruik te maken van OpenGL-functionaliteit ook enkele aliasing-effecten verwijderen: met de ingebouwde functies `glPointSmooth` en ook `glLineSmooth` kunnen getekende punten en (belangrijker voor deze implementatie) lijnen worden voorzien van een basis anti-aliasing. Verdere verbeteringen met bijvoorbeeld `glBlend` zijn mogelijk, maar introduceren een grotere performantie-afname voor een eerder minimale verbetering van de weergavekwaliteit. Voor meer technische details wordt verwezen naar de broncode.



Figuur 4.17: Verbetering temporele coherentie door het filteren van microcontouren en fading. **Links:** Algoritme 4.3. **Rechts:** Algoritme 4.3 aangevuld met temporele coherentie-verbeteringen uit secties 4.6.1 - 4.6.5. De  $t_d$  waarde gebruikt voor de figuren aan de rechterzijde is telkens 0.01.

#### 4.6.6 Resultaten / Besluit

Een vergelijking van enkele testobjecten zonder en mét de temporele coherentie-verbeteringen uit secties 4.6.1 - 4.6.5 wordt weergegeven in figuur 4.17. Het is duidelijk te zien hoe (vooral bij de laatste figuur) storende micro-contouren deels worden weggefilterd door de extra limiet op  $D_w \kappa_r$ , en deels quasi-onzichtbaar gemaakt worden door het *fading*-schema..

Een mogelijke verdere verbetering voor temporele coherentie zou erin bestaan om de snelheid van suggestieve contouren toch in rekening te brengen, door eventueel in een precomputatiestap per gebied de richting te bepalen waarop de gradiënt van  $\kappa_r$  snel van grootte verandert, om deze vervolgens als gewichten in te brengen bij het aanmaken van een contoursegment.

### 4.7 Aanvulling: Suggestieve highlights

De mathematische structuren opgebouwd voor suggestieve contouren in sectie 4.3 geven aanleiding tot een duale set contourlijnen: Suggestieve highlights. Deze lijnen worden in deze sectie kort gesitueerd, gedefinieerd en besproken.

#### 4.7.1 Situering

In lijntekeningen is het met de technieken die tot nu toe besproken werden in secties 4.2 en 4.3 mogelijk om de outline van een object weer te geven, alsook de locaties waar zich een vallei bevindt. Men wilt echter ook heuvels - bijvoorbeeld een elliptische kromming naar de camera toe - kunnen aanduiden met een type lijn [Lee et al., 2007].

Ook vormt het gebruik van bepaalde shadingtechnieken in combinatie met het tekenen van (suggestieve) contourlijnen een probleem: bij een toon shader kunnen bijvoorbeeld grote stukken van het object in een donkere kleur gerenderd worden, waardoor de suggestieve contourlijnen verloren gaan. Simpelweg de suggestieve contouren in een lichte kleur weergeven geeft hier geen coherent resultaat: er is een nieuw type lijnen nodig.

In [DeCarlo and Rusinkiewicz, 2007] worden hiervoor *suggestieve highlights* geïntroduceerd, een natuurlijke aanvulling op suggestieve contouren. Waar suggestieve contouren in een lijntekening de donkere delen van een (diffuus gerenderd) object abstraheren (te zien op figuur 4.6 rechts), zullen suggestieve highlights de helderste delen van het object samenvatten in contourlijnen.

Deze helder getinte lijnen zijn minder frequent terug te vinden in lijntekeningen: vele artiesten gebruiken witte kalk om grote vlakken wit te kleuren, maar zelden om curvatuur aan te duiden. Er zijn echter uitzonderingen: de theorie van suggestieve highlights werd ontwikkeld met het comicbook werk van Frank Miller [Miller, 2002] (figuur 4.18) en de schilderijen en zeefdrukken van Roy Liechtenstein in het achterhoofd.



*Figuur 4.18: Suggestieve highlights in de comicbook-kunst: Frank Miller, Sin City ©, 1991. Links: 'A Dame to Kill For'. Rechts: 'The Big Fat Kill'.*

#### 4.7.2 Theorie

Dat suggestieve highlights een natuurlijke aanvulling vormen op suggestieve contouren is eenvoudig in te zien aan de hand van de definitie: het zijn de overblijvende delen van de lussen van radiale curvatuur  $\kappa_r = 0$ , namelijk waar  $D_{\mathbf{w}}\kappa_r$  kleiner is dan 0:

$$\kappa_r = 0 \text{ en } D_{\mathbf{w}}\kappa_r < 0 \quad (4.18)$$

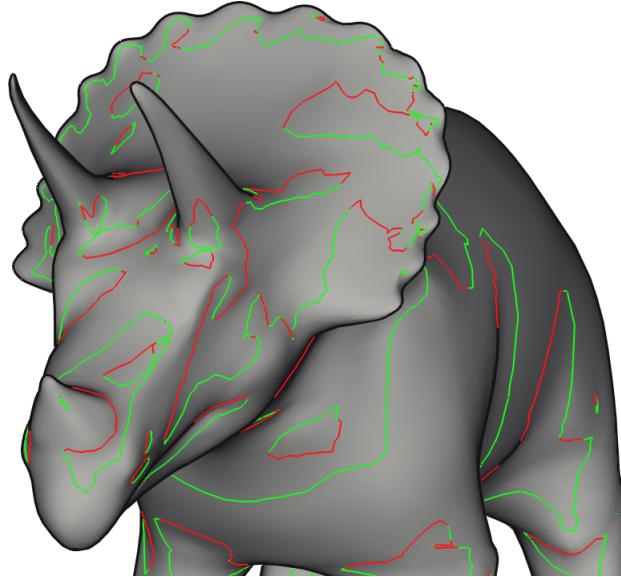
In figuur 4.19 wordt de dualiteit van beide soorten contourlijnen gedemonstreerd.

#### 4.7.3 Implementatie en evaluatie

De implementatie van (4.18) is triviaal: in algoritme 4.3 worden de ongelijkheidstesten op  $D_{\mathbf{w}}\kappa_r$  simpelweg omgedraaid. Ook dit geeft weer aanleiding tot geen, één of twee suggestieve highlightsegmenten per vlak, afhankelijk van het aantal nulpunten van  $D_{\mathbf{w}}\kappa_r$ . Er moet bij een eventuele single-pass aanpak van het tekenen van suggestieve contouren én suggestieve highlights rekening mee worden gehouden dat beiden simultaan op hetzelfde vlak kunnen voorkomen - dit volgt uit de duale definitie van de twee soorten contourlijnen.

De performantieverbeteringen uit sectie 4.4 zijn eveneens toepasbaar op suggestieve highlights. Aangezien deze contourlijnen bestaan uit punten waar  $\kappa_r = 0$  kunnen zij eveneens niet voorkomen op vlakken met een positieve Gaussiaanse curvatuur  $K$ . De precomputatie-filtering van vlakken uit sectie 4.4.1 is dus toepasbaar. Het negeren van onzichtbare suggestieve highlights met backface culling uit sectie 4.4.2 is ook onveranderd toepasbaar op suggestieve highlights.

Door deze overeenkomsten met suggestieve contouren werden suggestieve highlights zonder een significante performantieverschil aan de bestaande CPU Object



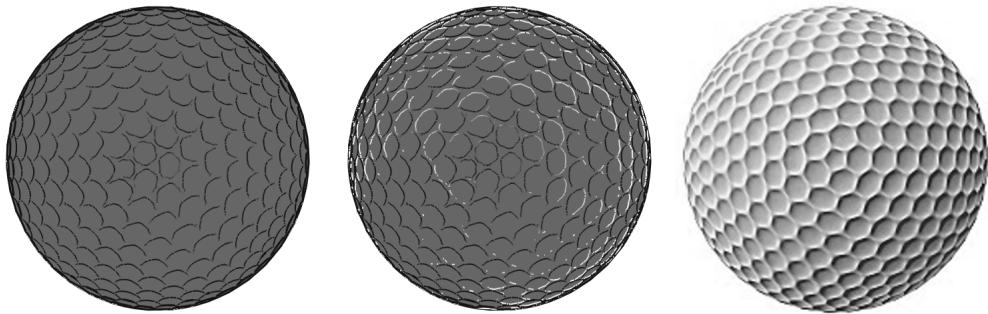
Figuur 4.19: *Suggestieve contouren* en *suggestieve highlights* vormen gezamenlijk de lussen waarop de radiale curvatuur  $\kappa_r = 0$ . Zo vormen suggestieve highlights een natuurlijke aanvulling op de suggestieve contouren.

Space implementatie toegevoegd. Het marginale performantieverschil is enkel te wijten aan de extra segmenten die nu moeten berekend en getekend worden.

Model	Gem. FPS (SC+C)	Gem. FPS (SH+SC+C)
Triceratops	65	62
Cow	27	24
Elephant	16	13
Armadillo	5	4
Heptoroid	3	2

Tabel 4.6: Performantievergelijking bij het uitvoeren van de benchmarks na het toevoegen van suggestieve highlights in een single pass, gezamenlijk met suggestieve contouren. Voor beide soorten contourlijnen werd algoritme 4.3 gebruikt, aangevuld met de performantie/temporele coherentie-verbeteringen uit de vorige secties.

Ook de temporele coherentieverbeteringen die werden ingevoerd voor gewone contourlijnen in sectie 4.6 kunnen met enkele lichte aanpassingen worden gebruikt voor suggestieve highlights. Het fadingschema uit sectie 4.6.4 wordt omgedraaid: de basiskleur van de contourlijnsegmenten is nu helder i.p.v. donker. Voor de gecombineerde test op de hoek  $\theta$  en  $D_w \kappa_r$  uit (4.15) wordt wel een afzonderlijke limiet gebruikt voor het filteren van suggestieve highlights.



Figuur 4.20: **Links:** enkel contouren en suggestieve contouren. **Midden:** Contouren, suggestieve contouren en suggestieve highlights geven een effect dat neigt naar ‘exaggerated shading’ (**Rechts**) uit [Rusinkiewicz et al., 2006], waarin de plaats van de lichtbronnen lokaal wordt aangepast om het grootst mogelijke shadingverschil te creëeren.

#### 4.7.4 Tekenstijlen

Er zijn enkele manieren om suggestieve highlights toe te voegen aan de bestaande contourlijnen om extra informatie over de curvatuur van het object weer te geven. Deze stijlen werden voorgesteld in [DeCarlo and Rusinkiewicz, 2007].

Algemeen kan opgemerkt worden dat suggestieve highlights veel minder breed inzetbaar zijn dan suggestieve contouren, en telkens met de nodige voorzichtigheid de juiste stijl moet gekozen worden voor de juiste toepassing: simpelweg alle bekende contourlijnen tekenen leidt niet noodzakelijk tot een ondubbelzinnige weergave van de curvatuur van het object.

##### C + SC + SH op donkere mesh

In deze stijl worden contouren en suggestieve contouren in het zwart getekend, suggestieve highlights in het wit, en de rest van het object in een donkergrizze kleur. Zo wordt een stijl bekomen die visueel erg overeenkomt met de ‘overdreven shading’ techniek uit [Rusinkiewicz et al., 2006]: de curvatuur van het object wordt met behulp van suggestieve highlights overgeaccentueerd, waardoor de krommingen in het oppervlak extra in de verf gezet worden. Een vergelijking wordt gemaakt in figuur 4.20.

Het tekenen van suggestieve contouren samen met suggestieve highlights kan echter ook een dubbelzinnige impressie geven van de curvatuur van het object, vooral als de volledige lussen van  $\kappa_r = 0$  duidelijk zichtbaar zijn vanuit een bepaald camerastandpunt: er wordt dan een foutief reliëf-effect gecreëerd (‘embossed’), zoals te zien is in figuur 4.21, vooral op de neus van het Cow-model.



Figuur 4.21: Indien zowel suggestieve contouren als suggestieve highlights getekend worden kunnen de lussen van de radiale curvatuur  $\kappa_r = 0$  een foutieve ‘relief’-impressie geven, hier duidelijk te zien op de neus van het Cow-model, die ‘ingedrukt’ lijkt te zijn.

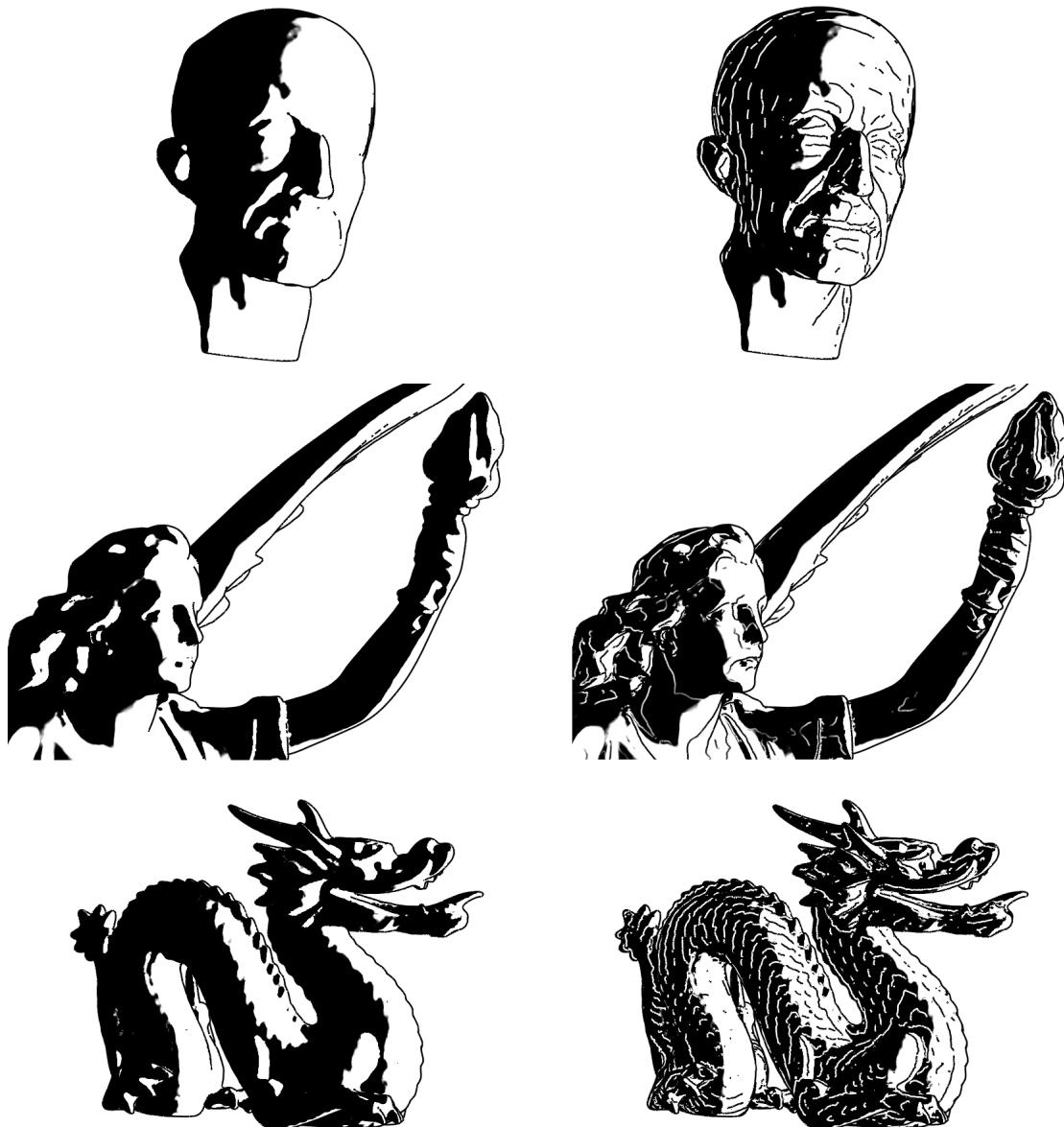
#### C + SC + SH met zwart/witte *toon shading*

*Toon shading* geeft een comicbook-achtig belichtingseffect aan de beschouwde objecten: het is een eenvoudige belichtingsmanier waarbij alle gebieden met een  $(\mathbf{n} \cdot \mathbf{l})$  waarde die kleiner is dan een bepaalde limiet (met  $n$  de eenheidsnormaalvector op het vlak en  $l$  de vector naar de lichtbron) een bepaalde kleur krijgen (zwart), en alle waarden boven de limiet een andere kleur (wit): er worden geen gradiënten getekend zoals bij diffuse belichting.

Indien zulke toon shading wordt toegepast op een object en vervolgens de suggestieve contouren (onzichtbaar op de zwarte stukken) en de suggestieve highlights (onzichtbaar op de witte stukken) getekend worden krijgt men een effect dat sterk gelijkt op de tekenstijl van Frank Miller [Miller, 2002], zoals in figuur 4.18. Op figuur 4.22 wordt dit effect gedemonstreerd voor enkele testmodellen.

#### 4.7.5 Besluit

Suggestieve highlights vormen een natuurlijke en eenvoudig implementeerbare aanvulling op suggestieve contouren. De definitie van deze contourlijnen werd erg geïnspireerd door de (comicbook) kunst. Hun toepassingsgebied is echter beperkter: deze lijnen zijn minder aanwezig in normale lijntekeningen, maar mits een juiste renderstijl kunnen ze van groot informatief en esthetisch nut zijn.



Figuur 4.22: Door het gebruik van een zwart-witte toon shading en het tekenen van suggestieve contouren en suggestieve highlights kan de tekenstijl van Frank Miller benaderd worden: de witte highlights geven in de zwart gekleurde vlakken de curvatuur van het object weer. **Links:** Het object gerenderd met de zwart-witte toon shading. **Rechts:** Het object na toevoeging van de suggestieve highlights en suggestieve contourlijnen.

## Hoofdstuk 5

# Object Space Algoritmes: GPU

In dit hoofdstuk worden eveneens Object Space algoritmes behandeld, net zoals in hoofdstuk 4. Nu wordt er in de voorgestelde algoritmes echter *wel* gebruik gemaakt van de functies die een hedendaagse GPU (Graphics Processing Unit) biedt. De implementatie en gevolgen van deze aanpak worden besproken en geëvalueerd.

### 5.1 Technologie en begrippen

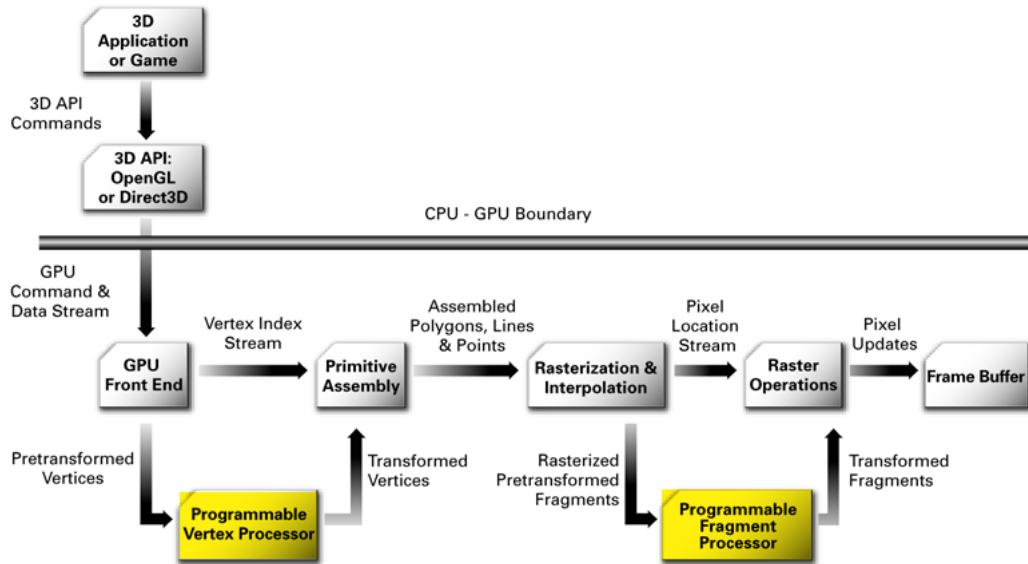
#### 5.1.1 De GPU: van *fixed function* naar programmeerbare pipeline

Voor een lange tijd was de GPU een *fixed-function-pipeline*: een ontwikkelaar kon enkel gebruik maken van de functionaliteit die standaard was ingebouwd in de grafische kaart, en was voor nieuwe functies afhankelijk van de stuurprogramma's van de fabrikant. Zo werd het moeilijk om nieuwe algoritmes efficiënt te kunnen uitvoeren met ondersteuning van de GPU. De algoritmes die besproken werden in hoofdstuk 4 zitten altijd met een spreekwoordelijk '*glazen plafond*' als het op performance op de CPU aankomt: er is nog ruimte voor optimalisaties, maar de snelheid van het uitvoeren van de aparte, atomische basisbewerkingen speelt een grote rol.

In 2001 bracht NVIDIA de eerste serie grafische kaarten op de consumentenmarkt die programmeerbare *shaders* hadden: kleine stukken programma die konden 'ingeplugged' worden in de bestaande pipeline om transformaties uit te voeren op data in de GPU: een nieuwe wereld voor graphics-gerelateerd programmeren ging open [Bailey and Cunningham, 2009]. In figuur 5.1 wordt een moderne GPU schematisch weergegeven.

Doorheen de jaren ontstonden er drie soorten shaders, elk met hun eigen functie. Ze worden hieronder besproken in normale volgorde van voorkomen:

- **Vertex shader:** kan operaties uitvoeren per vertex: het aanpassen van de positie, normaalvector, kleur of textuur coördinaat, maar ook ingewikkeldere per-vertex berekeningen. De uitvoer van de vertex shader gaat naar de geometry shader.
- **Geometry shader:** kan vertices toevoegen en weghalen uit de mesh: deze soort shader wordt bijvoorbeeld gebruikt om proceduraal extra detail toe te



Figuur 5.1: Een moderne GPU pipeline (zonder geometry processing). De gemarkeerde stukken zijn relevant voor het gebruik van shaders.

voegen aan bestaande meshes. De uitvoer van de geometry shader gaat naar de rasterizer, een component in de pipeline die de polygonen beschreven in de mesh omzet naar de uiteindelijke pixels waaruit het gerenderde beeld zal bestaan.

- **Pixel/Fragment shader<sup>1</sup>:** berekent de kleur van individuele pixels, met behulp van de input van de rasterizer. Waarden die berekend werden per vertex worden automatisch geïnterpolateerd over de huidige polygon. De uitvoer van de pixel shader gaat naar de framebuffer, die ervoor zorgt dat het gerenderde beeld uiteindelijk op het scherm verschijnt.

Door gebruik te maken van deze shaders kunnen we de redeneringen achter de Object Space algoritmes uit hoofdstuk 4 opnieuw maken en op een erg efficiënte manier contourlijnen berekenen en tekenen met behulp van de GPU.

### 5.1.2 Implementatiedetails

#### CPU-gedeelte

Hoewel de focus in dit hoofdstuk op het gebruik van de aanwezige grafische hardware ligt, is er implementatiegewijs nog steeds een CPU-applicatie nodig die de testmodellen inleest, de juiste data verplaatst, de shaders inlaadt en de uiteindelijke

<sup>1</sup> Noot: OpenGL Gebruikt de term *Fragment Shader*, Direct3D de term *Pixel Shader*. Deze laatste term wordt als foutief beschouwd, gezien er geen één-op-eén relatie is tussen het aantal oproepen van de shader en pixels op het scherm: sommige shaders worden meermaals per pixel uitgevoerd.

resultaten op het scherm kan tonen. Hiervoor werd opnieuw gebruik gemaakt van een C++/OpenGL-combinatie met dezelfde libraries zoals beschreven in sectie 4.1, aangevuld met de `TextFile` library<sup>2</sup>.

De benchmarkmethode die wordt toegepast voor de performantietests in dit hoofdstuk is identiek aan de werkwijze beschreven in sectie 4.1.4. Er wordt eveneens van dezelfde testmodellen gebruik gemaakt.

### High-Level Shadingtaal: GLSL

De eerste shaders werden geschreven in een assembly-taal, maar na enkele jaren werden er ook verschillende high-level programmeertalen ontwikkeld om shaders gemakkelijk te kunnen definiëren. In deze masterproef wordt gebruik gemaakt van de *OpenGL Shading Language* (GLSL), beschikbaar in OpenGL 1.5 en hoger [Khronos, 2009]. Hiervoor werd gekozen omdat de CPU-implementatie reeds was uitgewerkt in OpenGL, en dus enige affiniteit met de terminologie en werking van deze API aanwezig was. Alternatieven waren Microsoft's *High Level Shading Language*(HLSL) of NVIDIA's *C for Graphics* (Cg), waarnaar de shadercode uit dit hoofdstuk gemakkelijk portable is, gezien deze talen voornamelijk in syntax van elkaar verschillen: de basisconcepten zijn dezelfde.

## 5.2 Taking it to the GPU: Voordelen / Nadelen

Het gebruik van shaders heeft als eerste voordeel dat de hardware van een GPU gebouwd en geoptimaliseerd is voor een groot aantal vector en -matrixbewerkingen, die allen in parallel kunnen worden uitgevoerd. Zo kunnen basisoperaties aan een hogere snelheid en met een grotere *throughput* worden uitgevoerd dan mogelijk is op de CPU. Zeker voor performantie-gerichte applicaties is dit een groot voordeel.

Een tweede voordeel ligt in de portabiliteit en combineerbaarheid van de code: aangezien shaders kleine ‘pluggable’ stukken code zijn kunnen ze naar hartenlust gecombineerd worden, en eenvoudig in en -uitgeschakeld worden *at runtime*. Het gebruik van shaders in interactieve toepassingen, en dan vooral in recente video games, heeft het realisme van deze weergaven enorm verhoogd: de algoritmes waren al langer bekend in de academische wereld, maar er was nog geen efficiënte manier om hun effect op een interactieve wijze te kunnen presenteren.

Een nadeel is dat bij het ontwikkelen van shaders er rekening mee moet gehouden worden dat normale debugging tools zoals een *state inspector*, *breakpoint detector* of zelfs eenvoudige *print statements* niet beschikbaar zijn. Men kan de GPU best beschouwen als een *black box*: shaders worden éénmaal gecompileerd en ingeplugged in de pipeline, en de enige feedback bestaat uit de uiteindelijke pixelkleuren in de framebuffer.

Dit impliceert ook dat figuren renderen met behulp van shaders vaak een *one-way* procedure is: het is wel mogelijk om berekende vertexinformatie terug in het systeemgeheugen te laden, maar in GLSL is dat erg omslachtig. Nieuwere GPU-talen

---

<sup>2</sup>TextFile Library (GPL) - <http://www.lighthouse3d.com/opengl/>

zoals CUDA en recentelijk OpenCL moeten deze kloof dichten: zij moeten van de GPU een GPGPU maken (*General Purpose GPU*), waardoor op een transparante manier parallelle taken van de CPU kunnen uitgevoerd worden op de GPU, zelfs al zijn die taken niet noodzakelijk van graphics-gerelateerde aard. Voor de implementaties gemaakt in het teken van deze masterproef zou het gebruik van CUDA/OpenCL echter weinig voordeel hebben opgeleverd: alle operaties die worden uitgevoerd op de GPU zijn immers *one-way* en erg graphics-specifiek.

Een tweede nadeel van het werken op de GPU is dat enkele concepten minder gemakkelijk of niet definieerbaar zijn op bepaalde plekken in de pipeline: indien met behulp van een fragment shader beslist moet worden of een bepaalde pixel al dan niet deel uitmaakt van een contourlijn, kunnen we ons niet meer beroepen op het concept ‘*lijn*’, omdat dit simpelweg niet meer beschikbaar is in dat stadium van de pipeline (zie figuur 5.1). Welke geometrische vormen ook gedefinieerd werden aan de hand van vertices in de originele mesh, na de rasterizing stap zijn deze herleid tot pixels met bijbehorende geïnterpoleerde vertexwaarden. Deze verschillen met een CPU-implementatie zorgen ervoor dat een GPU-implementatie vanuit een andere invalshoek moet bekijken worden.

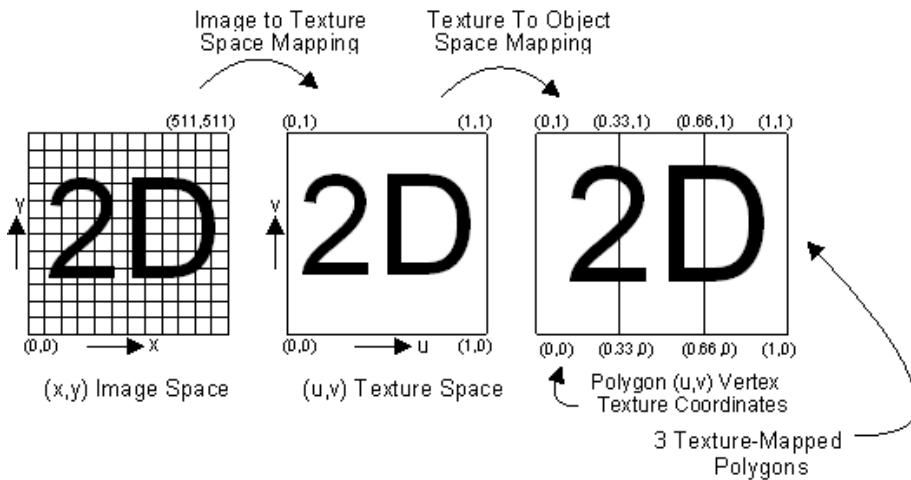
### 5.3 Suggestieve contouren via *Texture Mapping*

Een eerste algoritme dat gebruik maakt van de functionaliteit die de GPU biedt wordt voorgesteld in [Decarlo et al., 2004]: door toepassing van *texture mapping* om de contourlijnen te tekenen kan een deel van de tests uit algoritme 4.3 impliciet worden uitgevoerd. De breedte van de contourlijnen kan onder controle gehouden worden door op verschillende *mipmap-niveaus* de lijnbreedte constant te houden. Om een beter inzicht te krijgen in dit algoritme worden deze twee belangrijke technieken kort uitgelicht.

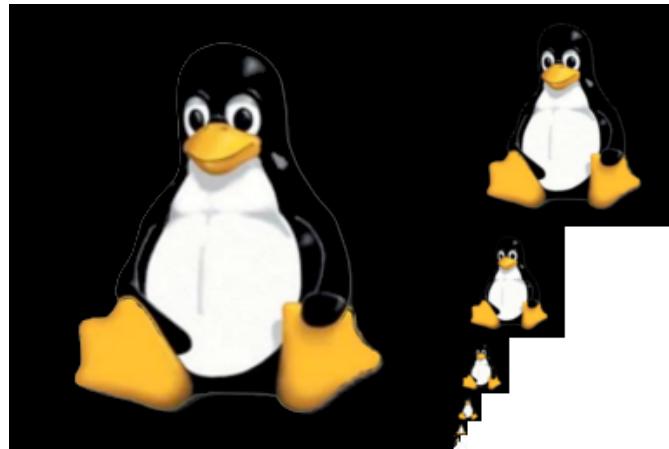
#### Texture Mapping

Bij texture mapping wordt aan elke vertex een textuurcoördinaat toegekend. Voor een 2D-map is deze van de vorm  $(u, v)$ . Deze textuurcoördinaat verwijst naar een positie op een *texture map*. Daarin is op elke positie een bepaalde kleur met rgb-waarden  $(r, g, b)$  opgeslagen. Deze kleuren kunnen handmatig in de texture map ingevoerd zijn of uitgelezen worden uit een bepaalde afbeelding.

Bij het renderen van het object wordt dus per positie op een polygon een interpolatie gemaakt van de textuurcoördinaten in de vertices. Deze  $(u_i, v_i)$  geven dan aanleiding tot een positie op de texture map, die als resultaat een kleur  $(r, g, b)$  teruggeeft. Deze kleur kan op verschillende manieren bekomen zijn: het eenvoudigste schema neemt de dichtstbijzijnde kleur in de texture map. Beter is bijvoorbeeld om het gemiddelde te nemen van de dichtstbijzijnde kleuren [Heckbert and Moreton, 1991]. Het texture mapping-proces wordt schematisch weergegeven in figuur 5.2.



Figuur 5.2: Texture Mapping proces. (Afbeelding Okino CG©)

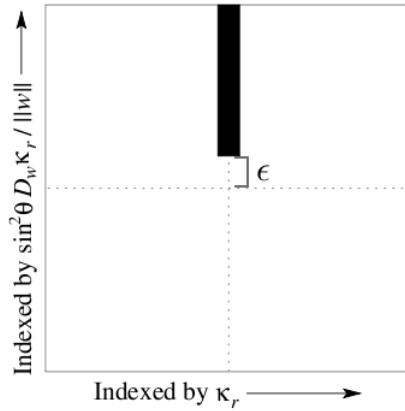


Figuur 5.3: Een textuur van  $128 \times 128$  texels en zijn mipmaps, in aflopende grootte.

### Mipmapniveaus

Deze techniek werd ontwikkeld door [Williams, 1983]. Het principe is om aan een basis-textuur een reeks geoptimaliseerde, kleinere versies van diezelfde textuur toe te voegen: dit zijn de zogenaamde *mipmaps*<sup>3</sup>. Een gehele *mipmap set* voor een textuur van grootte  $n \times n$  heeft  $\log_2 n$  mipmaps. Een textuur van  $128 \times 128$  texels heeft dus 7 mipmaps: een mipmap van  $64 \times 64$ ,  $32 \times 32$ ,  $16 \times 16$ ,  $8 \times 8$ ,  $4 \times 4$ ,  $2 \times 2$  en van  $1 \times 1$  texels. De extra opslagruimte nodig om deze mipmap set op te slaan is een derde van de opslagruimte voor de originele textuur. Praktisch worden mipmaps opgeslagen zoals schematisch wordt weergegeven in figuur 5.3.

<sup>3</sup>In ‘mipmap’ is ‘MIP’ een acroniem van het Latijnse ‘*Multum in Parvo*’, wat vertaalt naar ‘veel in weinig’.



*Figuur 5.4: De textuurmap die gebruikt wordt om suggestieve contouren te tekenen via texture mapping. Door  $\kappa_r$  en de limiettest op  $D_w \kappa_r$  als indices te kiezen kan met deze texturemap een lijn getekend worden. (Figuur gebaseerd op [Decarlo et al., 2004])*

Welke mipmap gebruikt wordt hangt af van het camerastandpunt: indien het camerastandpunt zich dicht bij het van textuur voorziene object bevindt, zal de mipmap met de hoogste resolutie gebruikt worden. Indien het camerastandpunt verder weggeplaatst wordt, zullen steeds mipmaps met een lagere resolutie gebruikt worden. Dit heeft twee voordelen: de performantie van de real-time rendering verbetert, gezien het werken met een kleiner mipmap minder rekentijd vraagt. Ook zijn de verschillende mipmaps reeds op voorhand ge-antialiased, wat het aantal artefacten bij het renderen verminderd zonder extra werk voor de renderer.

### 5.3.1 Algoritme: Contourlijnen via texture mapping

In [Decarlo et al., 2004] wordt voorgesteld om een texture map te genereren met een zwarte strook van vaste grootte. Deze strook wordt voor elk mipmapniveau even breed gehouden - de textuur wordt dus niet simpelweg geschaald zoals in de mipmaps op figuur 5.3. Door voor elk mipmap-niveau een unieke map te maken waarin de lijnbreedte steeds dezelfde blijft, probeert men in de uiteindelijke rendering op alle zoom-niveaus dezelfde lijndikte te hebben voor de contourlijnen.

De texturemap wordt geïndexeerd door enerzijds de radiale curvatuur  $\kappa_r$  en anderzijds de gecombineerde test op  $D_w \kappa_r$  die werd opgesteld in sectie 4.6.3. De textuur die wordt opgesteld is afgebeeld op figuur 5.4.

Er zal dus een zwarte lijn getekend worden indien  $\kappa_r$  een waarde dicht bij het nulpunt heeft en  $\sin^2 \theta \frac{D_w \kappa_r}{\|w\|}$  groter is dan een vooraf ingestelde limiet  $\epsilon$ . We moeten deze limiet wel nog vermenigvuldigen met de *feature size* van het object om  $\epsilon$  onafhankelijk te maken van de grootte van het gebruikte model (zie sectie 4.6.2).

Codeblok 5.1: Suggestieve contouren via Texture Mapping (Pseudo Code)

```

int TEXMAPSIZE = 128; // default textuurmap grootte in texels
int LINEWIDTH = 8, // default lijnbreedte in pixels
int mplevel 0;

// genereer texture map op elk mipmap-niveau
while(TEXMAPSIZE>0)
{
    generateTexture(TEXMAPSIZE,LINEWIDTH);
    storeTexture(mplevel);
    texmapsize=texmapsize/2;
    mplevel++;
}

// bereken radiale curvatuur en sin(theta)dwkr/|w| voor alle vertices
computeVertexInfo(kr,dwkrtest);
// zet textuurcoordinaten voor alle vertices
for(int i = 0; i < themesh->vertices.size(); i++)
{
    v.setTextureCoordinate2D(feature_size*kr[i];
    feature_size*feature_size*dwkrtest[i]);
}
// teken object
drawMesh();

```

### 5.3.2 Implementatie

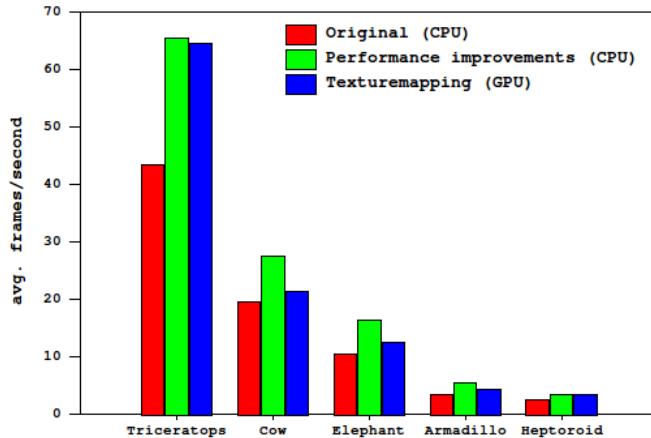
De implementatie van het algoritme is vrij *straightforward* en terug te vinden in Pseudo Code in het codeblok 5.1. Eerst worden de texturen aangemaakt op alle nodige mipmapniveaus: er blijven texturen aangemaakt worden tot het huidige mipmapniveau 1 is. Vervolgens worden de waarden voor de radiale curvatuur  $\kappa_r$  en  $D_w\kappa_r$  berekend, die dan als textuurcoördinaten voor de vertices worden ingesteld. Vervolgens kan de figuur getekend worden en doet de aanwezige graphics hardware (impliciet) de rest: de kleur voor elke pixel wordt gekozen uit de texture map.

### 5.3.3 Evaluatie

#### Performantie

De textuurcoördinaten in combinatie met de aangemaakte texture map zorgen ervoor dat de tests om nulpunten te vinden op een segment niet meer hoeven uitgevoerd te worden: dit gebeurt nu impliciet bij het texture mappen. Wel belangrijk is het feit dat het (computationeel) zware rekenwerk om voor elke vertex  $\kappa_r$  en  $D_w\kappa_r$  te berekenen nog steeds voor elke frame op de *CPU* moet gebeuren: enkel de tests per vlak en het selecteren van de kleur zijn impliciet verplaatst naar de GPU.

Daarom is de performantiewinst bij het gebruik van dit algoritme niet overweldigend. Zoals op figuur 5.5 te zien is, is de performantie bij het tekenen van contouren



Figuur 5.5: Het algoritme 5.1 voor het tekenen van contourlijnen vergeleken met de CPU-algoritmes uit hoofdstuk 4. Alle geteste algoritmes zijn multi-pass: contourlijnen en suggestieve contourlijnen worden in een afzonderlijke stap getekend.

en suggestieve contouren in twee *passes* beter dan voor het originele CPU algoritme 4.3, maar ongeveer even goed of zelfs slechter vergeleken met het CPU algoritme met de toegevoegde performantieverbeteringen uit sectie 4.4: de texture lookup zorgt voor nieuwe overhead. Wel kan men stellen dat algoritme 5.1 eenvoudiger is qua implementatie dan de voorgaande CPU-algoritmes.

### Correctheid

Indien men het resultaat van algoritme 5.1 bekijkt wordt meteen duidelijk aan welke prijs het uitbesteden van de testen op  $\kappa_r$  en  $D_{W\kappa_r}$  aan texture mapping komt: de controle over de lijndikte van de segmenten is moeilijk te behouden, zelfs bij het gebruik van een hoog aantal mipmaps. In figuur 5.6 is duidelijk te zien hoe op het Lucy-model verschillende lijndiktes verschijnen en in de close-up onderaan hoe de techniek van texture-mapping artefacten kan creëren door de textuurcoördinaten per vertex te interpoleren: de lijnen krijgen een houtskool-achtige stijl. Het probleem van variërende lijndikte is eigen aan Image Space algoritmes, zoals werd vermeld in 3.2.1, maar kan - zoals hier gedemonstreerd - ook een probleem vormen in Object Space algoritmes.

Belangrijk is ook om te vermelden dat het concept van contour *segmenten* hier verloren gaat: algoritme 5.1 heeft na een renderingstap geen lijst met geometrische punten die aangeven uit welke lijnstukken (segmenten) de gevonden contourlijnen bestaan: bij de Object Space CPU-algoritmes was dit wel het geval. Hierdoor wordt het toepassen van geavanceerde technieken (bvb. volgen van contourlijnsegmenten over de tijd) of het verkrijgen van statistieken over de contourlijnsegmenten bemoeilijkt of onmogelijk gemaakt.



*Figuur 5.6: Het texture-mapping algoritme 5.1 om contouren en suggestieve contouren te tekenen heeft als grote nadeel dat de controle over de lijndikte verloren gaat: de texture-mapping interpolatie kan ook artefacten introduceren, zoals gedemonstreerd in de close-up onderaan.*

### Temporele coherentie

In algoritme 5.1 wordt impliciet een vorm van fading toegepast, zoals werd voorgesteld in sectie 4.6.4: indien een punt een textuurcoördinaat heeft die op de randen van de zwarte strook in de texture-map ligt, zal de kleur automatisch geïnterpoleerd worden. De temporele coherentie van de gegenereerde beelden is aanvaardbaar.

#### 5.3.4 Besluit

Er kan besloten worden dat algoritme 5.1 tegenover de daling in complexiteit weinig verbeteringen stelt. De performantie is immers niet beter dan die van het verbeterde CPU algoritme uit sectie 4.4: enkel de tests op  $\kappa_r$  en  $D_w\kappa_r$  worden impliciet uitgevoerd op de GPU, via een (relatief dure) texture lookup. Het toepassen van deze texture mapping brengt eveneens het probleem van de variërende lijndikte met zich mee.

## 5.4 Suggestieve contouren via *Shaders*

In deze sectie wordt een **nieuw** GPU-gebaseerd Object Space algoritme geïntroduceerd, dat gebruik maakt van shaders om de berekeningen en limiet-testen uit te voeren op de GPU. Dit algoritme werd gedefinieerd en geïmplementeerd in het kader van deze masterproef. Wegens het duale karakter van suggestieve contouren met suggestieve highlights kan het ook gebruikt worden om deze highlights, besproken in sectie 4.7, te berekenen en te tekenen.

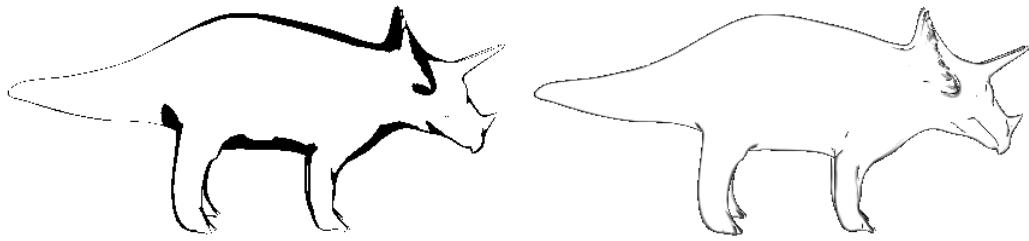
### 5.4.1 Theorie en begrippen

#### Gebruik van een vertex shader voor $\kappa_r$ en $D_w\kappa_r$

Een probleem in algoritme 5.1 was dat de berekening van  $\kappa_r$  en  $D_w\kappa_r$  nog steeds moest worden uitgevoerd op de CPU. Deze berekening is per frame echter alleen afhankelijk van de huidige view vector  $\mathbf{v}$ . De andere vereiste waarden (posities, normalen, principale richtingen, ...) kunnen éénmalig op voorhand berekend worden. Bovendien zijn deze berekeningen per vertex onafhankelijk van elkaar, en kunnen ze bijgevolg per frame uitgevoerd worden in een vertex shader, die als invoer slechts de huidige camerapositie vereist. Zo wordt gebruik gemaakt van de optimale ondersteuning voor vector-berekeningen en verregaande parallelisatie in de GPU.

#### Gebruik van fragment shader om contourlijnen te tekenen

In algoritme 5.1 werd de lijndikte bepaald door de verschillende mipmaps: voor een bepaalde pixel werd geïnterpoleerd tussen de verschillende waarden in de texture map: dit gaf aanleiding tot een sterk variërende lijndikte. Om dit te verhelpen zal afgestapt worden van texture-mapping interpolatie om deze lijndikte te bepalen, en zullen in deze sectie andere tests worden ingevoerd om te bepalen of een bepaalde pixel deel uitmaakt van een contourlijnsegment. Ook dit proces is onafhankelijk van de andere pixels: deze tests worden bijgevolg uitgevoerd in een fragment shader.



*Figuur 5.7: Links:* Normale contourlijnen detecteren per pixel met limiet (5.1). *Rechts:* Met limiet (5.2). Het is duidelijk te zien hoe zonder rekening te houden met de radiale curvatuur  $\kappa_r$  een limiet  $\epsilon$  onvoldoende is om overal voor een gelijke lijndikte van de contourlijnen te zorgen.

### Nieuwe limiet voor normale contourlijnsegmenten

Voor segmenten van gewone contourlijnen wordt volgens de definitie van de Contour Generator (2.1) op  $\mathbf{n} \cdot \mathbf{v} = 0$  getest: aangezien er in een gediscretiseerd domein van pixels gewerkt wordt is een eerste voorstel om  $\mathbf{n} \cdot \mathbf{v}$  te evalueren voor het geometrische punt waarnaar de pixel verwijst en te kijken of deze waarde in een vastgelegd interval  $\epsilon$  ligt:

$$|\mathbf{n} \cdot \mathbf{v}| < \epsilon \quad (5.1)$$

Dit zorgt echter voor brede lijnen in gebieden waar  $\mathbf{n} \cdot \mathbf{v}$  slechts traag rond het nulpunt varieert, zoals te zien is op figuur 3.2. Een betere test brengt dus ook de *kromming* of radiale curvatuur van het oppervlak in rekening:  $\kappa_r$  wordt toch sowieso berekend voor de test op suggestieve contourlijnen. In het kader van deze masterproef stellen we als nieuwe limiet om te beslissen of een bepaalde pixel  $\mathbf{p}_i$  tot een normale contourlijn behoort de limiet  $t_c$  voor:

$$\frac{(\mathbf{n} \cdot \mathbf{v})^2}{\kappa_r} < t_c \quad (5.2)$$

De waarde van  $\mathbf{n} \cdot \mathbf{v}$  wordt hierin gekwadrateerd om met een enkele threshold te kunnen werken. Het voorkomen van  $\kappa_r$  in de noemer van de limiet kan geïnterpreteerd worden als: indien  $\mathbf{n} \cdot \mathbf{v}$  groot is en men dus verder van het nulpunt vandaan zit, moet de radiale curvatuur  $\kappa_r$  van het oppervlak ook groot zijn om nog te kunnen behoren tot een contourlijnsegment. Een visuele vergelijking tussen het gebruiken van limiet (5.1) en de nieuwe limiet (5.2) wordt gemaakt op figuur 5.7.

### Nieuwe limiet voor suggestieve contourlijnsegmenten

Ook voor suggestieve contourlijnsegmenten moet een nieuwe limiet geformuleerd worden om per pixel te kunnen beslissen of de pixel  $\mathbf{p}_i$  in kwestie tot de suggestieve contourlijn behoort. Volgens de definitie van de Suggestieve Contour Generator 2.3 wordt op  $\kappa_r$  getest: aangezien er in een gediscretiseerd domein van pixels gewerkt wordt is een eerste voorstel dus om de radiale curvatuur te evalueren en te kijken of de



*Figuur 5.8: Links: detecteren van suggestieve contourlijnen per pixel met limiet 5.3. Rechts: Met limiet 5.4. Indien er geen rekening wordt gehouden met de snelheid waarmee  $\kappa_r$  varieert is te zien hoe suggestieve contourlijnen worden 'uitgesmeerd' over grotere gebieden, met zeer storende vlekken als resultaat.*

waarde binnen een domein  $\epsilon$  rond het nulpunt ligt (aangevuld met de convexiteitstest op  $D_{\mathbf{w}}\kappa_r$ ):

$$|\kappa_r| < \epsilon \text{ en } D_{\mathbf{w}}\kappa_r > 0 \quad (5.3)$$

Het toepassen van deze limiet geeft echter een analoog probleem aan het toepassen van limiet 5.1 bij normale contourlijnsegmenten: gebieden waar de radiale curvatuur  $\kappa_r$  slechts traag varieert zullen dikkere suggestieve contourlijnen bevatten. Om de snelheid in variatie van  $\kappa_r$  in rekening te brengen kunnen we  $D_{\mathbf{w}}\kappa_r$  gebruiken. Deze wordt gebruikt in de nieuwe voorgestelde limiet  $t_{sc}$ . De limiet  $t_{dwkr}$  wordt vervolgens ingevoerd om numerieke instabiliteiten in de berekening van  $D_{\mathbf{w}}\kappa_r$  tegen te gaan:

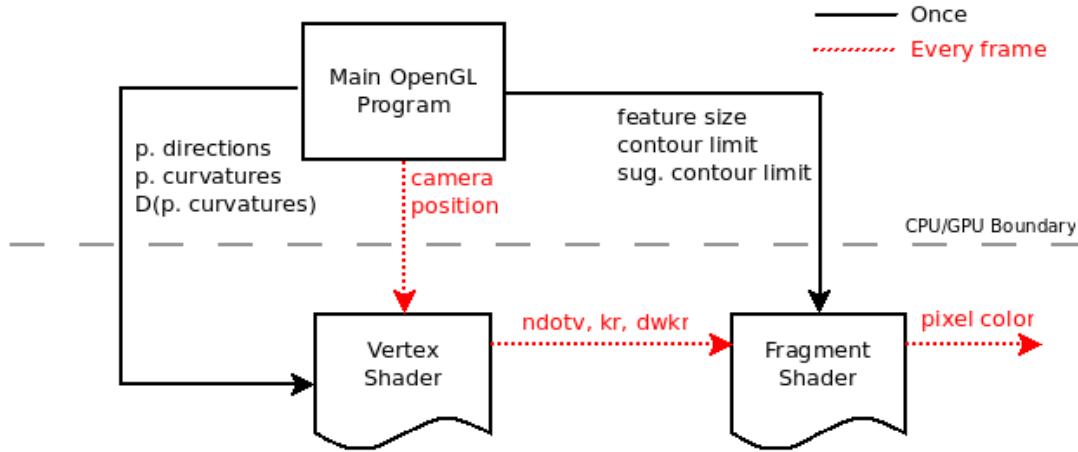
$$\frac{|\kappa_r|}{D_{\mathbf{w}}\kappa_r} < t_{sc} \text{ en } D_{\mathbf{w}}\kappa_r > t_{dwkr} \quad (5.4)$$

Het voorkomen van  $D_{\mathbf{w}}\kappa_r$  in de noemer van 5.4 kan geïnterpreteerd worden als volgt: indien  $\kappa_r$  relatief groot is en men dus relatief ver verwijderd is van een lus die een nulpunt vormt van de radiale curvatuur, moet de afgeleide van de radiale curvatuur in de richting van de vector  $\mathbf{w}$  ook groot zijn om nog te kunnen behoren tot een suggestief contourlijnsegment. Een visuele vergelijking tussen het gebruik van limiet 5.3 en de nieuwe limiet 5.4 wordt weergegeven op figuur 5.8.

Door het invoeren van deze twee nieuwe limieten (die zich enkel baseren op data die reeds beschikbaar was in de vorige algoritmes) kunnen we de lijnbreedte beter onder controle houden zonder daarbij te moeten inboeten aan performantie: er zijn geen computationeel dure berekeningen vereist.

### Fading

Ook op de nieuwe limieten 5.2 en 5.4 kan een eenvoudig fading-algoritme worden toegepast: hoe verder de waarden voor een bepaalde pixel  $\mathbf{p}_i$  positief van deze



Figuur 5.9: Opbouw van het GPU-shader algoritme om contourlijnen te tekenen. De vertex shader heeft per frame enkel het nieuwe camerastandpunt nodig om de waarden voor  $n \cdot v$ ,  $\kappa_r$  en  $D_w\kappa_r$  te berekenen.

limieten verwijderd zijn, hoe donkerder de pixel getekend wordt. Dit gebeurt op een analoge manier als in het fading schema uit 4.6.4.

#### 5.4.2 Implementatie

In deze sectie worden de nieuwe technieken uit sectie 5.4.1 vertaald in een bruikbaar algoritme. Bij de beschrijving van de implementatie van dit algoritme is het gebruik van enkele specifieke GLSL-concepten onvermijdelijk. Voor meer informatie wordt verwezen naar de GLSL Reference Manual [Khronos, 2009].

In figuur 5.9 wordt een algemeen overzicht gegeven van de *flow* van de variabelen tussen de verschillende elementen in het algoritme. De 3 belangrijkste elementen (CPU programma, vertex shader en fragment shader) worden elk afzonderlijk besproken in de volgende secties.

#### CPU gedeelte

Enkele waarden kunnen éénmalig voor elk model op voorhand berekend worden in een zogenaamde *pre-process* stap. In deze masterproef wordt niet ingegaan op de verschillende manieren waarop deze waarden kunnen bekomen worden: hiervoor zijn enkele standaardalgoritmes beschikbaar, bijvoorbeeld [Taubin, 1995]. De volgende (camerastandpunt-onafhankelijke) vertexeigenschappen worden gekend verondersteld:

- Principale richtingen  $e_1$  en  $e_2$  en bijhorende curvaturen  $\kappa_1$  en  $\kappa_2$  (per vertex)
- Afgeleiden van beide principale curvaturen in beide principale richtingen:  $D_{e_1}\kappa_1, D_{e_2}\kappa_1, D_{e_1}\kappa_2$  en  $D_{e_2}\kappa_2$  (per vertex)
- Feature size (per model, zie sectie 4.6.2)

Codeblok 5.2: S. Contouren via GPU-shader - CPU gedeelte (Pseudo Code)

```
// bereken waarden voor model
compute_principal_curvatures(mesh);
compute_dcurv(mesh);
compute_feature_size(mesh);
// laad shaders
loadShaders();
// draw loop
while(draw)
{
    // camerapositie doorgeven aan shaders
    setCameraPosition(camera);
    // teken object
    drawMesh(mesh);
}
unloadShaders();
```

Deze waarden zullen eenmalig op de CPU berekend worden en blijven voor de rest van het algoritme dezelfde: er wordt gesteld dat deze overal beschikbaar zijn, zoals aangegeven met de zwarte lijnen op figuur 5.9.

Om in de vertex shader  $\kappa_r$  en  $D_w\kappa_r$  te kunnen berekenen, zijn voor elke vertex de positie  $\mathbf{p}_i$ , de eenheidsnormaalvector  $\mathbf{n}_i$  en het huidige camerastandpunt  $\mathbf{c}$  vereist. De eerste twee waarden zijn impliciet beschikbaar in de GPU-pipeline, de camerapositie zal per frame manueel moeten worden doorgeven, zoals aangegeven wordt op figuur 5.9 met de rode lijnen. Dit gebeurt via een `uniform` variabele: in GLSL betekent dit dat de waarde van de variabele niet zal veranderen tijdens de `drawMesh()`-call. Dit is correct: voor elke vertex die wordt ‘bezocht’ in hetzelfde frame zal de camera op dezelfde absolute positie staan. De code voor het CPU-gedeelte wordt in Pseudo Code gegeven in codeblok 5.2.

### Vertex Shader

In de vertex shader wordt  $\kappa_r$  berekend met formule van Euler (4.1) en  $D_w\kappa_r$  met de afgeleide formule (4.11). Aangezien de waarden van  $\mathbf{n} \cdot \mathbf{v}$ ,  $\kappa_r$  en  $D_w\kappa_r$  beschikbaar moeten zijn voor interpolatie in de fragment shader, worden zij opgeslagen in zogenaamde `varying` variabelen: hiermee geeft men in GLSL aan dat waarden interpoleerbaar zijn per pixel. Verder wordt in de vertex shader nog de perspectief en view-transformatie uitgevoerd: deze matrixvermenigvuldigingen gebeuren met het commando `ftransform()`. De GLSL-code voor de vertex shader wordt gegeven in codeblok 5.3.

Hierin werd onmiddellijk een kleine optimalisatie doorgevoerd: indien een bepaalde vertex  $\mathbf{v}_i$  onzichtbaar is vanuit het huidige camerastandpunt ( $\mathbf{n} \cdot \mathbf{v} < 0.0$ ) worden de overige waarden niet berekend: deze zullen toch niet gebruikt worden.

Codeblok 5.3: S. Contouren via GPU-shader - Vertex Shader (GLSL)

```
// IN: camerapositie (per frame)
uniform vec3 cam_pos;
// OUT: waarden voor fragment shader
varying float ndotv,kr;dwkr;
void main()
{
    // view vector berekenen
    vec3 view = cam_pos - vec3(gl_Vertex.x,gl_Vertex.y,gl_Vertex.z);
    // n.v berekenen
    ndotv = (1.0f / length(view)) * dot(gl_Normal,view);
    // als het vertexpunt zichtbaar is, ga verder
    if(!(ndotv < 0.0f)){
        // radiale curvatuur berekenen (formule 4.1)
        vec3 w = normalize(view - gl_Normal * dot(view, gl_Normal));
        float u = dot(w, pdir1); float u2 = u*u;
        float v = dot(w, pdir2); float v2 = v*v; float uv = u*v;
        kr = (curv1*u2) + (curv2*v2);
        // dw_kr berekenen (formule 4.11)
        float dwII = (u2*u*dcurv.x) + (3.0*u*uv*dcurv.y)
            + (3.0*uv*v*dcurv.z) + (v*v2*dcurv.w);
        dwkr = dwII + 2.0 * curv1 * curv2
            * ndotv/sqrt((1.0 - pow(ndotv, 2.0)));
    }
    // perspectief/view transformatie
    gl_Position = ftransform();
}
```

## Fragment Shader

In de fragment shader worden de waarden die berekend werden in de vertex shader (impliciet) geïnterpolateerd en vergeleken met de ingestelde limieten. Aan de hand van het fading schema wordt vervolgens de pixelkleur gekozen, en opgeslagen in `gl_FragColor`, de GLSL-variabele die op het einde van de fragment shader de finale pixelkleur moet bevatten. De GLSL-code voor de fragment shader wordt weergegeven in codeblok 5.4.

### 5.4.3 OpenGL/GLSL-specifieke aanpassingen

Om de performantie van het GPU-shader algoritme uit sectie 5.4.2 te garanderen, werden enkele OpenGL en GLSL-specifieke aanpassingen doorgevoerd. Deze worden wegens hun beperkte algemene karakter slechts kort besproken.

- **Vertex Buffer Objects:** Voor een correct gebruik van shaders hoeft het originele model slechts op zeer eenvoudige wijze getekend te worden: alle (mogelijk computationeel zware) belichting die met behulp van OpenGL op

Codeblok 5.4: S. Contouren via GPU-shader - Fragment Shader (GLSL)

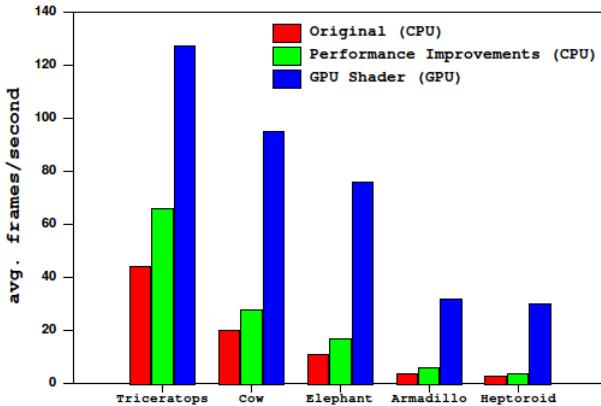
```
// IN: waarden van vertex shader
varying float ndotv, kr, dwkr;
// IN: feature size en limieten
uniform float fz, t_c, t_sc, t_dwkr;
void main()
{
    // basiskleur
    vec4 color = vec4(1.0f, 1.0f, 1.0f, 1.0f);
    // vermenigvuldigen met feature size
    float kr = fz * kr;
    float dwkr = fz * fz * dwkr;
    // limieten berekenen
    float c_limit = t_c*(pow(ndotv, 2) / kr);
    float sc_limit = t_sc*(abs(kr) / dwkr);
    // contouren
    if(c_limit < 1.0)
        color.xyz = min(vec3(c_limit, c_limit, c_limit), color.xyz);
    // suggestieve contouren
    else if((sc_limit < 1.0) && (dwkr > t_dwkr))
        color.xyz = min(vec3(sc_limit, sc_limit, sc_limit), color.xyz);
    gl_FragColor = color;
}
```

de CPU gebeurt zal toch overschreven worden met waarden uit de fragment shader.

Daarom worden alle vertexcoördinaten en eenheidsnormaalvectoren in een *Vertex Buffer Object* geladen en eenmalig in het geheugen van de GPU geplaatst [Ho, 2005]. Zo volstaat het om bij elke render het model gewoon fullbright (= wit) te tekenen.

- **Textuurcoördinaten:** Om de verschillende voorberekende waarden uit sectie 5.4.2 beschikbaar te maken in de vertex shader wordt gebruik gemaakt van een kleine ‘truc’: al deze waarden worden als *textuurcoördinaat* opgeslagen in een afzonderlijk Vertex Buffer Object, en zo impliciet beschikbaar gesteld in de shader. Zo zijn er geen extra calls nodig om deze waarden per frame door te sturen naar de GPU. Wel zijn voor deze methode minstens 5 *texture units* vereist: dit is gelukkig geen probleem op de meeste moderne GPU hardware.
- **GLSL instructieoptimalisatie** Hoewel GLSL al enkele jaren beschikbaar is als High-Level shader programmeertaal, is het belangrijk om te begrijpen hoe de instructies vertaald worden door de GLSL compiler. Zo werd erop gelet om formules te definiëren in de MAD-vorm<sup>4</sup>: dit kan op de GPU uitgevoerd worden in 1 cyclus, waar een andere - mathematisch equivalente - formulering extra cycli in beslag zou nemen.

<sup>4</sup>MAD - *Multiply, then Add*



Figuur 5.10: Vergelijking van het GPU-shader algoritme om contourlijnen en suggestieve contourlijnen te tekenen uit sectie 5.4.2 met het originele CPU algoritme uit sectie 4.3 en het verbeterde CPU algoritme uit sectie 4.4. Het GPU-shader algoritme kent een aanzienlijk hogere performantie dan de CPU-gebaseerde algoritmes.

#### 5.4.4 Evaluatie

##### Performantie

Door zowel de berekening van  $\kappa_r$  en  $D_w\kappa_r$ , alsook de limiet-evaluatie te laten uitvoeren op de GPU wordt optimaal gebruik gemaakt van de efficiënte vectorbewerkingen en verregaande parallelisatie die een hedendaagse GPU biedt. Dit is ook te merken in de vergelijkende performantiestatistieken, weergegeven in figuur 5.10: het GPU-shader algoritme levert voor alle testmodellen framerates die voldoende zijn voor interactieve toepassingen.

Vooral voor grotere meshes wordt de verbetering die het GPU-shader algoritme biedt duidelijk: bij een klein model zoals de Triceratops (22 460 polygonen) is de verbetering tegenover het originele CPU algoritme uit sectie 4.3 ongeveer een drievoud, bij een groot model zoals de Heptoroid (573 440 polygonen) is een verbetering merkbaar van ongeveer een *twaalfvoud*.

##### Correctheid

Het GPU-Shader algoritme biedt wegens de in sectie 5.4.1 ingevoerde limieten  $t_c$  en  $t_{sc}$  een goede controle over de lijndikte van de getekende contourlijnen. Door het vermenigvuldigen van  $\kappa_r$  en  $D_w\kappa_r$  met de *feature size* kan bovendien met eenzelfde limietwaarde over de hele figuur eenzelfde lijndikte bekomen worden. Enkele resultaten worden weergegeven op figuur 5.11.

De goede werking van de limieten  $t_c$  en  $t_{sc}$  is vooral merkbaar bij het Hersenenmodel, meer bepaald aan het onderste deel van de hersenlobben, waar een fijne lijnstructuur gecreëerd wordt met de suggestieve contourlijnen.

Nadeel van dit algoritme is natuurlijk wel dat er per frame geen lijst beschikbaar is met lijnstukken die de segmenten van de contourlijnen vormen: dit bijhouden

strookt niet met het opgebouwde shader-framework en de manier waarop de GPU pipeline is opgebouwd (zie sectie 5.1).

Omdat in het GPU-shader algoritme pas per *pixel* beslist wordt of een punt al dan niet behoort tot een contourlijnsegment wint men aan rendersnelheid, maar verliest men aan informatiewaarde omdat op dat punt in de pipeline abstractie gemaakt is van de originele geometrie van het object. Dit doet misschien denken aan een Image Space algoritme, maar merk op dat we nog steeds berekeningen uitvoeren met de originele data van de mesh, en onze contourlijnen baseren op geïnterpoleerde waarden van de oorspronkelijke vertices.

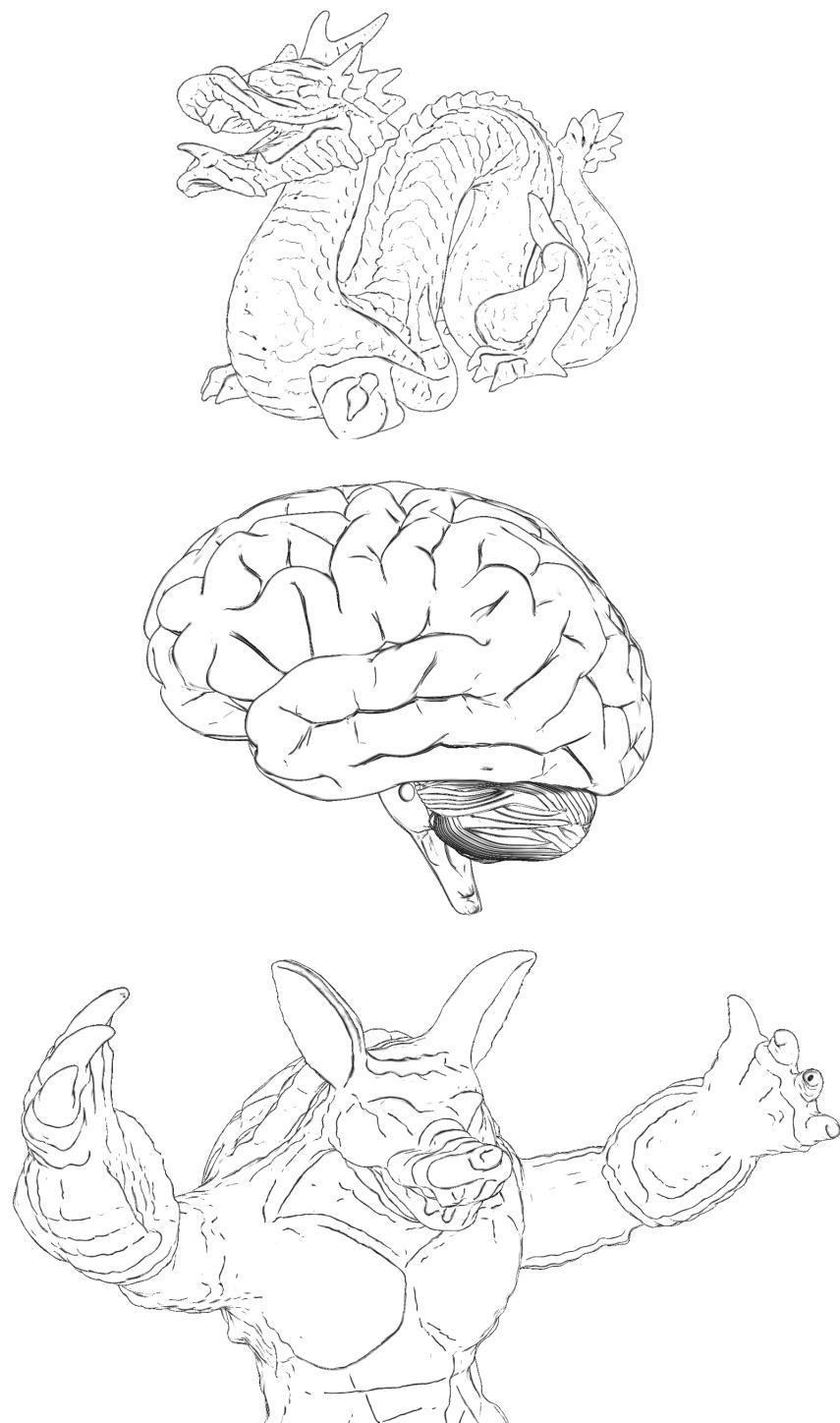
### Temporele coherentie

Omdat voor de fading van de suggestieve contouren nu gebruik wordt gemaakt van de afstand tot de limieten  $t_c$  en  $t_{sc}$  in plaats van de texture-mapping interpolatie uit sectie 5.3 zijn er weinig of geen storende artefacten in het finale beeld aanwezig. De limiet op  $D_w \kappa_r$  slaagt er ook in om storende microcontouren weg te filteren, zodat de temporele coherentie van de renders hoog is.

#### 5.4.5 Besluit

Het in het kader van deze masterproef ontwikkelde nieuwe GPU-shader algoritme uit sectie 5.4.2 blinkt uit in performantie: zelfs voor de complexe modellen uit de benchmark worden met onze implementatie framerates neergezet die bruikbaar zijn in *real-time* toepassingen.

In ruil hiervoor verliest men de mogelijkheid om alle contoursegmenten geometrisch vast te leggen, alsook de precieze controle over de lijndikte. Door middel van het invoeren van de nieuwe limieten  $t_c$  en  $t_{sc}$  om te beslissen of een pixel al dan niet tot een contourlijnsegment behoort kunnen we dit opvangen en de lijndikte beter regelen dan in het reeds bestaande Texture Mapping-algoritme 5.1.



*Figuur 5.11: Resultaten van het tekenen van contouren en suggestieve contourlijnen (in dezelfde pass) met het GPU-shader algoritme uit sectie 5.4.2.*

# Hoofdstuk 6

## Image Space Algoritmes

In dit hoofdstuk worden Image Space algoritmes behandeld. Deze algoritmes worden gekenmerkt door een andere opbouw dan de tot hiertoe behandelde Object Space algoritmes uit hoofdstukken 4 en 5. Verschillende invalshoeken om tot een performant en correct Image Space algoritme te komen worden voorgesteld, vergeleken en geëvalueerd.

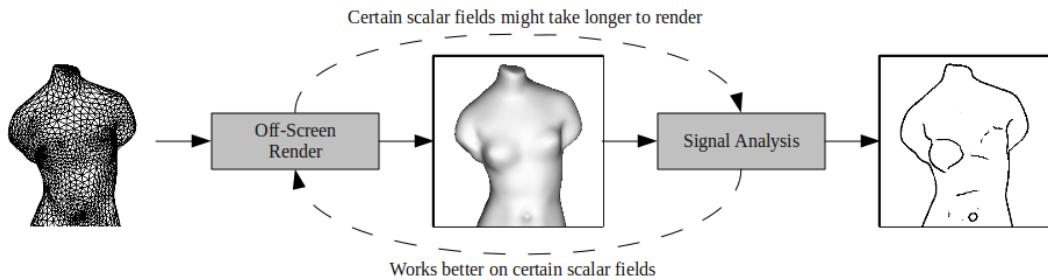
### 6.1 Technologie en begrippen

#### 6.1.1 Image Space: Een andere aanpak

In sectie 3.1 werd de structuur van een Image Space algoritme besproken: er wordt eerst een scalair veld naar keuze gerenderd, waarop vervolgens een signaalverwerkingsmethode wordt toegepast. Concreet komt het er op neer dat een model dat voorzien moet worden van (suggestieve) contourlijnen eerst *off-screen* zal gerenderd worden, en op dit gerenderde 2D-beeld vervolgens operaties zullen worden toegepast om uiteindelijk de contourlijnen te extraheren. Deze contourlijnen kunnen vervolgens over een bepaalde voorstelling van het gewenste model getekend worden door middel van een eenvoudige convolutie. Dit proces wordt schematisch weergegeven op figuur 3.1.

#### 6.1.2 Rendercomplexiteit vs. Analysecomplexiteit

Om een Image Space algoritme voor contourlijnextractie op te stellen zijn er twee elementen die van belang zijn: de off-screen rendering stap en de signaalverwerkingsmethode op dat resultaat. Deze zijn echter niet los van elkaar te koppelen: bepaalde signaalverwerkingsmethoden verwachten een bepaalde input om een nuttige uitvoer te kunnen bieden. Het genereren van een bepaald soort scalair veld vertrekende van het model kan dan weer weinig of meer rekenwerk vragen: beide stappen in een Image Space algoritme zijn altijd onmiskenbaar met elkaar in balans. Hierdoor komt het ook dat Image Space algoritmes altijd minstens twee *passes* kennen. Een schematische voorstelling van deze wisselwerking is te vinden op figuur 6.1.



Figuur 6.1: De verschillende stappen in een Image Space algoritme.

### 6.1.3 Signaalverwerking via fragment shaders

Aangezien het scalair veld waarop gewerkt wordt in dit geval een rooster van pixels is (de tussentijdse render) en de signaalverwerkingsmethoden enkel werken op deze pixelinformatie is het aangewezen om deze methodes te implementeren in afzonderlijke fragment shaders. Hiermee kunnen eenvoudig algoritmes gedefinieerd worden die per pixel de uiteindelijke kleur kunnen beslissen - hiervoor is telkens het volledige scalair veld gevormd door de pixelwaarden van de tussentijdse render beschikbaar.

Affiniteit met deze manier van werken, en meer specifiek de High-Level Shading Language GLSL, werd aangebracht tijdens de implementatie van de algoritmes uit hoofdstuk 5. De GLSL-implementaties die gemaakt werden in het kader van dit hoofdstuk zijn echter eenvoudig portable naar vergelijkbare shadertalen.

### 6.1.4 Overige software en benchmarkmethode

In dit hoofdstuk werd het omliggende framework voor de algoritmes wederom geïmplementeerd met behulp van C++ en OpenGL (zie sectie 4.1 voor details). Om enkele testscenes te modelleren werd gebruik gemaakt van het modelleerpakket Blender.

De benchmarkmethode om de performantie van de voorgestelde algoritmes te meten is dezelfde zoals beschreven in sectie 4.1.4, met een aanvulling: aangezien Image Space algoritmes zoals vermeld in sectie 3.2.4 impliciet rekening houden met view-dependent *level-of-detail* is de resolutie waarop de tests worden uitgevoerd nu ook van belang: daarom zal de performantie op dezelfde wijze ook gemeten worden op hogere resoluties.

## 6.2 Contouren via Threshold filter

Een eerste algoritme om normale contourlijnen te extraheren voor een bepaald model wordt voorgesteld in [Rusinkiewicz, 2008] en is rechtstreeks gebaseerd op de definitie van de Contour Generator (2.1).

### 6.2.1 Theorie

De definitie van de Contour Generator uit [Decarlo et al., 2003] definieert contourlijnen als verzamelingen van punten waar  $\mathbf{n} \cdot \mathbf{v} = 0$ . Door als scalair veld een render te nemen van de  $\mathbf{n} \cdot \mathbf{v}$ -waarden van het model kan per pixel een eenvoudige threshold filter worden toegepast: als de waarde klein genoeg is wordt de pixel tot een contourlijn gerekend. Het meeste rekenwerk situeert zich in deze werkwijze dus duidelijk bij het renderen van het scalair veld.

#### Diffuse Belichting

Om dit scalair veld te renderen kan gebruik gemaakt worden van *diffuse belichting*, het eenvoudigste belichtingsmodel (zie bijvoorbeeld [Shirley et al., 2005]) dat ondersteund wordt in OpenGL en nagenoeg elke andere moderne Graphics API. De intensiteit van een punt in deze belichtingsmethode wordt bepaald door  $\mathbf{n} \cdot \mathbf{l}$ , waarbij  $\mathbf{n}$  de normaalvector voorstelt in dat punt en  $\mathbf{l}$  de vector in de richting van de lichtbron. Indien de lichtbron op dezelfde locatie wordt geplaatst als de camera is  $\mathbf{l} = \mathbf{v}$  en krijgt men het gewenste scalaire veld. Deze techniek werd ook toegepast voor het creëren van contourlijnen voor cartoons in [Paerson and Robinson, 1985].

#### Threshold filter

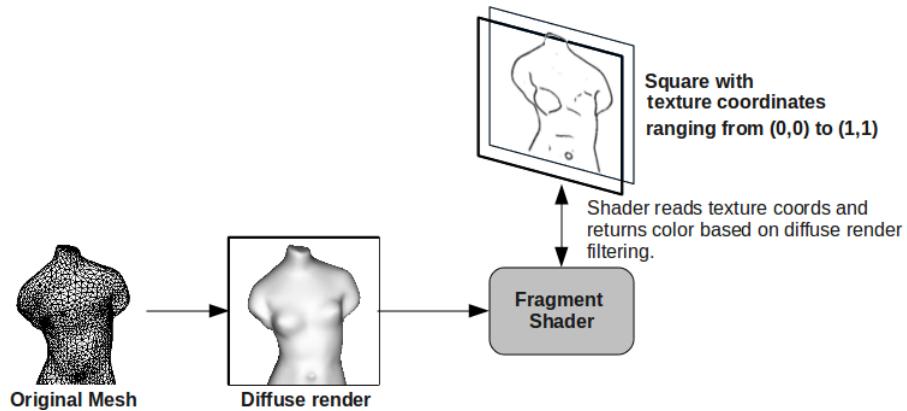
De signaalverwerkingsmethode die wordt toegepast op het gerenderde beeld is triviaal: indien de intensiteit van een bepaalde pixel onder de limiet  $t_c$  komt, krijgt deze pixel een donkere kleur: deze behoort tot de gezochte contourlijn.

### 6.2.2 Implementatie

De implementatie werd als volgt praktisch uitgewerkt in OpenGL/GLSL: per frame wordt een diffuus gerenderd beeld van het model weggeschreven naar een 2D textuur. Vervolgens wordt een vierkant getekend met breedte van de gewenste uitvoer. Dit vierkant heeft textuurcoördinaten die uniform verdeeld zijn tussen 0 en 1. Bij het tekenen van dit vierkant wordt de shader gebruikt die voor elke pixel de textuurkleur bepaalt op basis van de texture van het diffuus gerenderde beeld. Dit proces wordt schematisch voorgesteld in figuur 6.2. De werkwijze per frame kan in Pseudo Code worden teruggevonden in codeblok 6.1.

De code van de shaders is eenvoudig. In de vertex shader wordt enkel de view en perspectieftransformatie uitgevoerd en worden de textuurcoördinaten voor gebruik op de texture map met de tussentijdse render beschikbaar gemaakt. De GLSL-code voor de vertex shader is terug te vinden in codeblok 6.2.

## Contouren via Threshold filter



Figuur 6.2: Door het renderen van een vierkant met uniform verdeelde texture-coördinaten kan men in algoritme 6.1 de resulterende kleur van de fragment shader op het scherm brengen.

Codeblok 6.1: Contouren via Threshold filter - CPU gedeelte (Pseudo Code)

```

int WIDTH = 512;
// plaats licht op positie van camera
addPointLight(cameraPosition, Color(1.0,1.0,1.0));
addLightModel(DIFFUSE);
texture diffuse;
// render object met diffuse belichting
disableBuffer(); // geen zichtbare output
drawMesh();
// sla resultaat op in texture
copyImageToTex(diffuse);
// kopieer texture naar shader
enableShader(threshold);
setDiffuseTexture(diffuse);
// teken textured vierkant met shader
enableBuffer();
drawQuad(WIDTH);

```

Codeblok 6.2: Contouren via Threshold filter - Vertex Shader (GLSL)

```

void main()
{
    // textuurcoördinaat kopiëren
    gl_TexCoord[0] = gl_MultiTexCoord0;
    // matrixtransformatie
    gl_Position = ftransform();
}

```

Codeblok 6.3: Contouren via Threshold filter - Fragment Shader (GLSL)

```

uniform sampler2D diffuse_render;
uniform float t_c;

// deze functie zet RGB-waarden om in het kwadraat van de intensiteit
float intensity(in vec4 color){
    return ((color.x*color.x)+(color.y*color.y)+(color.z*color.z));}

void main(void)
{
    gl_FragColor = vec4(1,1,1,0); // achtergrondkleur
    // is de intensiteit van deze pixel kleiner dan de limiet t_c?
    if(intensity(texture2D(diffuse_render,gl_TexCoord[0].st)) < (t_c*t_c)){
        gl_FragColor = vec4(0,0,0,0); // pixel is deel van contour
    }
}

```

In de fragment shader wordt de waarde van de huidige pixel vergeleken met de threshold  $t_c$ . De intensiteit wordt gemeten als de som van de kwadraten van de RGB-bijdragen in de pixel. Om geen vierkantswortel (een dure berekening) te moeten evalueren wordt de limiet  $t_c$  eenvoudigweg gekwadrateerd. De GLSL-code voor de fragment shader wordt in codeblok 6.3 weergegeven.

### 6.2.3 Evaluatie

#### Performantie

De performantie van algoritme 6.1 is zeer goed: zelfs voor de grote modellen uit de benchmark wordt een framerate bereikt die interactieve toepassingen mogelijk maakt. Hiervoor zijn twee elementen verantwoordelijk. Ten eerste is het maken van de tussentijdse render zeer efficiënt, gezien het diffuse belichtingsmodel een eenvoudig model is dat goed ondersteund wordt in moderne graphics hardware en API's. Ten tweede is het werk dat gebeurt in de fragment shader erg eenvoudig: er moet slechts 1 waarde uitgerekend en getest worden. Resultaten van de benchmark op enkele testobjecten zijn terug te vinden in tabel 6.1.

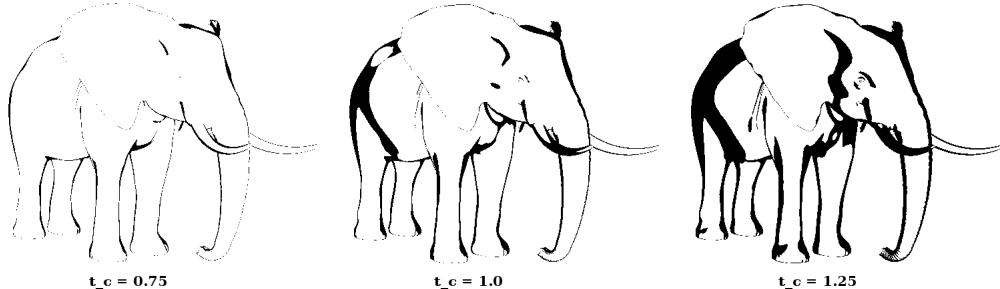
Interessant is om te kijken waar het verschil in framerate tussen de verschillende modellen vandaan komt: de tussentijdse renderstap vergt meer tijd voor grotere modellen - er zijn immers meer evaluaties nodig van  $n \cdot l$  om tot de correcte pixelkleur te komen. Dit effect is te zien in tabel 6.2. De bijkomende lichte stijging van rekentijd in de signaalanalysestep is enkel te wijten aan het feit dat er op meer vertices een matrixtransformatie moet uitgevoerd worden in de vertex shader 6.2: de rekentijd in de fragment shader is voor alle modellen dezelfde, gezien in alle gevallen op een scalair veld met eenzelfde grootte gewerkt wordt.

Model	Polycount	Gem. FPS
Triceratops	22 460	126
Cow	92 864	104
Elephant	157 160	74
Armadillo	345 944	22
Heptoroid	573 440	31

Tabel 6.1: Resultaten van benchmarks met algoritme 6.1, uitgevoerd op een resolutie van  $512 \times 512$  pixels.

Model	Diffuse render	Signaalverwerking
Triceratops	753	356
Cow	1417	486
Elephant	2783	534
Armadillo	13 130	753
Heptoroid	12 456	720

Tabel 6.2: Gemiddelde verwerkingsstijd (in  $10^{-6}$  seconden) van de twee passes in algoritme 6.1.

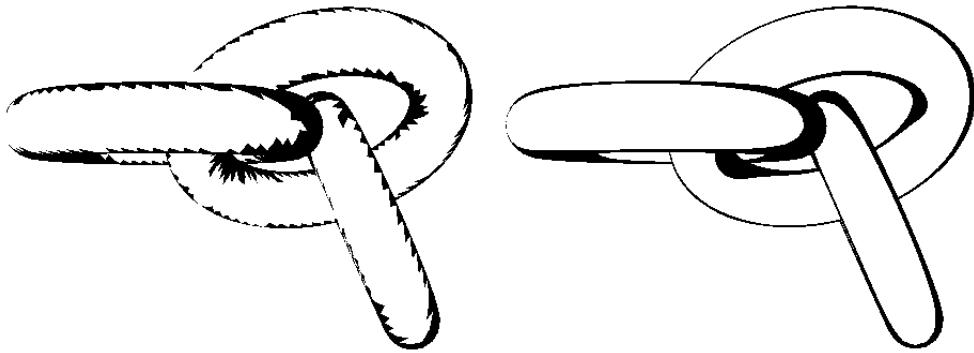


Figuur 6.3: Resultaten van algoritme 6.1, toegepast op het Elephant-testobject met verschillende waarden voor de limiet  $t_c$ . Het is onmogelijk om een waarde  $t_c$  te vinden waarvoor alle contourlijnen dezelfde dikte hebben.

### Correctheid

Het gebruiken van een enkele threshold  $t_c$  zorgt voor problemen: op plaatsen waar  $\mathbf{n} \cdot \mathbf{v}$  slechts licht varieert worden de contourlijnen duidelijk dikker: dit verlies van controle over lijndikte is een typisch verschijnsel in Image Space algoritmes: het is onmogelijk om een waarde  $t_c$  te vinden waarvoor alle contourlijnen dezelfde dikte hebben. Dit wordt gedemonstreerd op figuur 6.3.

In [Rusinkiewicz, 2008] wordt voorgesteld om dit probleem aan te pakken door niet  $\mathbf{n} \cdot \mathbf{v}$  te renderen als scalar veld, maar  $\frac{\mathbf{n} \cdot \mathbf{v}}{\kappa_r}$ . Hiervoor is echter een extra berekening



Figuur 6.4: Resultaten van algoritme 6.1, met een diffuse render die voorzien werd van flat shaded diffuse belichting (*links*) en van Gouraud shaded diffuse belichting (*rechts*).

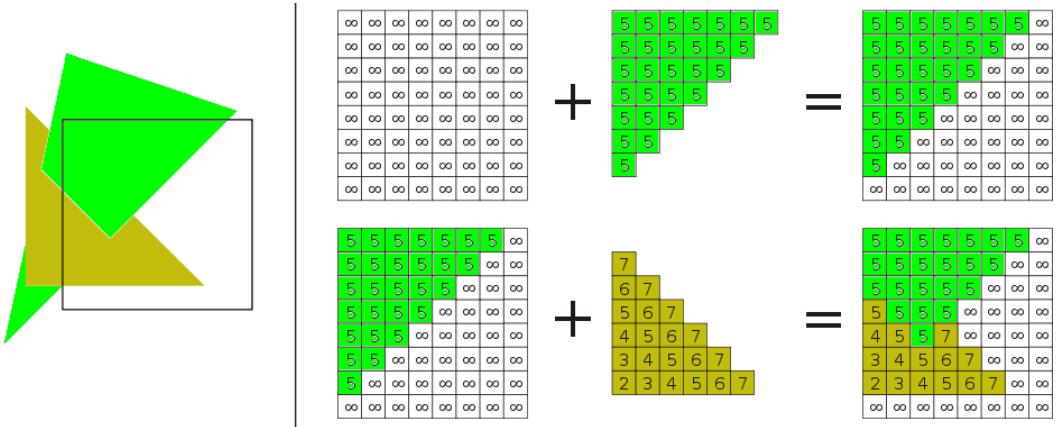
nodig in Object Space: de radiale curvatuur  $\kappa_r$  zou dan bepaald moeten worden met behulp van formule (4.1). Hierdoor zou dit algoritme geen ‘(pure)’ Image Space methode meer zijn. Bovendien werd een vergelijkbare methode al met succes toegepast in het GPU-shader algoritme uit sectie 5.4.2, en zou er in dit geval een groot verlies aan nauwkeurigheid zijn door de resultaten te mappen op een RGB-kleurenruimte, waar in de fragment shader 5.4 van het GPU-shader algoritme kan gerekend worden met de originele waarde voor  $\kappa_r$ . Omwille van deze redenen zal in deze masterproef niet worden ingegaan op deze suggestie.

### Temporele coherentie

De temporele coherentie van het renderen van modellen met algoritme 6.1 hangt onder meer af van de kwaliteit van de tussentijdse diffuse rendering. Indien geen gebruik wordt gemaakt van *Gouraud shading* [Gouraud, 1971] (ook wel *smooth shading* genoemd) maar van simpele *flat shading* krijgt men in plaats van een vloeiende evolutie van lijndikte plotselveranderingen van lijnstijl en zijn de polygonen ook zeer duidelijk zichtbaar, wat de contourlijnen een sterk gekarteld uiterlijk geeft. Een vergelijking wordt gemaakt op figuur 6.4.

#### 6.2.4 Besluit

Algoritme 6.1 is eenvoudig en performant, maar wegens het gebrek aan controle over de lijndikte, gedemonstreerd in figuur 6.3, zullen met deze methode geen resultaten kunnen bereikt worden vergelijkbaar met die van de Object Space algoritmes uit hoofdstuk 4 en 5.



Figuur 6.5: Het opbouwen van de *z-buffer* bij het tekenen van meerdere polygonen.

## 6.3 Contouren via Sobel filter

Een tweede algoritme combineert een complexere signaalverwerkingsmethode met een eenvoudigere tussentijdse rendering: door het toepassen van een *Sobel filter* kunnen we in een *depth map* randen detecteren. Deze twee concepten worden kort toegelicht.

### 6.3.1 Theorie

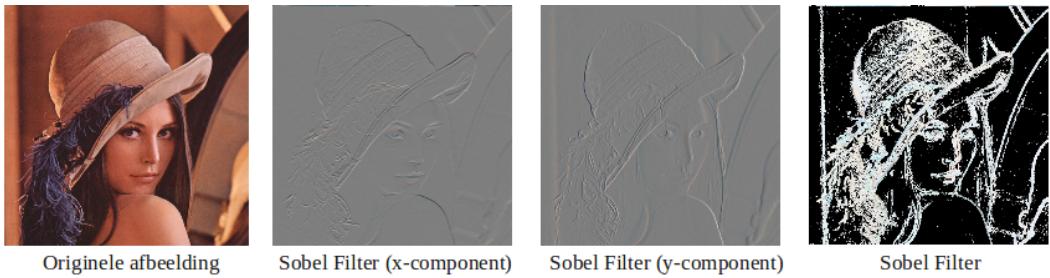
#### Depth Map

Als scalair veld wordt in het Sobel filter-algoritme gebruik gemaakt van een *depth map*. Dit is een structuur waarin voor elke pixel  $\mathbf{p}_i$  de diepte of z-waarde wordt bijgehouden: hoe hoger de z-waarde, hoe verder de locatie die de huidige pixel voorstelt van de camera verwijderd is. De *depth buffer* of *z-buffer* wordt gebruikt om bij het tekenen van meerdere objecten de visibiliteit van een object te checken. Indien de z-waarde kleiner blijkt te zijn dan die in de *depth buffer* hoeft dat deel van het object in kwestie niet getekend te worden: het zit verborgen achter een ander object. Dit principe wordt geïllustreerd in figuur 6.5.

#### Sobel Filter

Om uit de depth map contouren te extraheren volstaat een simpele thresholding-aanpak niet: er moet nu aan randdetectie gedaan worden. Interessant is hoe deze methode om contourlijnen te extraheren het werk in de andere schaal legt van de balans die werd besproken in sectie 6.1.2: de tussentijdse rendering is eenvoudiger, maar de signaalverwerkingsmethode die wordt uitgevoerd op het gerenderde beeld is complexer.

Een mogelijke methode om aan randdetectie te doen is de zogenaamde Sobel Filter. Deze basistechniek, voorgesteld in [I. Sobel, 1968], maakt gebruik van een discrete differentiatie-operator om een schatting te maken van de gradiënt in een



Figuur 6.6: Het toepassen van de Sobel Filter beschreven in sectie 6.3.1 op een afbeelding. Door het samenvoegen van de horizontale component  $G_x$  en de verticale component  $G_y$  bekomt men het uiteindelijke gefilterde resultaat  $G$ , uiterst rechts op de afbeelding.

punt van een scalair veld opgebouwd uit de intensiteiten van de afbeelding. Het resultaat is slechts een ruwe benadering van de gradiënt, maar is voor afbeeldingen zonder veel hoogfrequente componenten nauwkeurig genoeg.

De discrete differentiatie-operator  $G$  is opgesplitst in twee componenten, die kunnen berekend worden met twee specifieke matrices met grootte 3x3, *kernels* genaamd. Deze moeten geconvolueerd worden met de afbeelding  $A$  om de horizontale component  $G_x$  en verticale component  $G_y$  te berekenen.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * A \text{ en } G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A \quad (6.1)$$

Deze componenten kunnen dan gecombineerd worden om tot de uiteindelijke schatting  $G$  van de grootte van de gradiënt in het punt te komen:

$$G = \sqrt{G_x^2 + G_y^2} \quad (6.2)$$

Met dezelfde componenten kan ook de richting  $\Theta$  van de gradiënt bepaald worden:

$$\Theta = \arctan \frac{G_y}{G_x} \quad (6.3)$$

Een voorbeeld van de Sobel Filter toegepast op een afbeelding is te vinden op figuur 6.6. Merk op dat de Sobel Filter-techniek veel ruis genereert in gebieden met hoogfrequente componenten, zoals bijvoorbeeld de hoed van de dame op de figuur.

Om de kernels te kunnen convolueren in uitdrukking (6.1) kunnen uiteraard enkel matrices gebruikt worden van grootte 3x3. Daarom zullen per punt in het scalaire veld slechts de 8 omliggende pixels aangevuld met de centrale pixel gebruikt worden om de gradiënt te schatten. Hogere-orde randdetectiemethoden maken gebruik van meer pixels om een betere schatting van de gradiënt te maken maar zijn veel complexer.

Codeblok 6.4: Contouren via Sobel filter - CPU gedeelte (Pseudo Code)

```

int WIDTH = 512;
disableBuffer(); // geen zichtbare output
addLightModel(NONE); // geen belichting
texture depth;
// render object om depth map op te bouwen
drawMesh();
// sla resultaat op in texture
copyDepthToTex(depth);
// kopieer texture met depth map naar shader
enableShader(sobel);
setDepthTexture(diffuse);
// teken textured vierkant met shader
enableBuffer();
drawQuad(WIDTH);

```

### Contouren via gradiënt

Door aan de met een Sobel Filter geschatte waarde voor de gradiënt een limietwaarde  $ts_c$  op te leggen kan men contourlijnen extraheren: het zijn punten waar de gradiënt groot is, en waar het scalaire veld dus grote veranderingen ondergaat in waarde. Dit zijn plaatsen op de depth map waar er wordt overgegaan van een locatie op het object zelf naar een locatie op oneindig - in praktijk is dit de maximum z-waarde die de hardware aankan. Op figuur 6.5 is dit effect te zien in de depth map, meer bepaald aan de randen van de niet-overlappende stukken van de groene en gele polygonen.

#### 6.3.2 Implementatie

De implementatie van het Sobel Filter-algoritme werd als volgt praktisch uitgewerkt in OpenGL/GLSL: per frame wordt het model gerenderd, waarbij enkel het opbouwen van de *depth map* uitgevoerd wordt: er wordt niets getekend, de pixelkleuren zullen uiteindelijk bepaald worden in de fragment shader. De waarden van deze depth map worden gekopieerd naar een textuur genaamd `depth`. Vervolgens wordt op het scherm een vierkant van gewenste zijde getekend met textuurcoördinaten tussen 0 en 1. Bij het tekenen van dit vierkant wordt de fragment shader gebruikt om de kleur van elke pixel te bepalen. In Pseudo Code wordt deze werkwijze voorgesteld in codeblok 6.4.

De implementatie van de vertex shader is triviaal en de GLSL-code is identiek aan die voor het Threshold-algoritme in codeblok 6.2. De perspectief en -viewtransformatie worden hierin uitgevoerd en de textuurcoördinaten worden beschikbaar gemaakt.

In de fragment shader wordt de Sobel Filter toegepast per pixel. Hiervoor worden uit de textuur `depth` de depth-waarden van de 8 omliggende pixels opgehaald. De resulterende schatting van de gradiënt  $G$  wordt vervolgens vergeleken met de ingestelde limietwaarde  $ts_c$ . Indien de gradiënt  $G$  voldoende groot is, wordt aan de

Codeblok 6.5: Contouren via Sobel filter - Fragment Shader (GLSL)

```

uniform sampler2D depth;
uniform float ts_c;
uniform int width;

float intensity(in vec4 color)
    return(color.x*color.x)+(color.y*color.y)+(color.z*color.z);

void main(void)
{
    gl_FragColor = vec4(1,1,1,0); // achtergrondkleur
    // de textuurcoordinaat voor pixel p
    vec2 p = gl_TexCoord[0].st;
    float s = 1.0/width; // step in uniforme sampler

    // waarden rond pixel opvragen
    float tl = intensity(texture2D(depth, p+vec2(-s,s)));
    float l = intensity(texture2D(depth, p+vec2(-s,0)));
    float bl = intensity(texture2D(depth, p+vec2(-s,-s)));
    float t = intensity(texture2D(depth, p+vec2(0,s)));
    float b = intensity(texture2D(depth, p+vec2(0,-s)));
    float tr = intensity(texture2D(depth, p+vec2(s,s)));
    float r = intensity(texture2D(depth, p+vec2(s,0)));
    float br = intensity(texture2D(depth, p+vec2(s,-s)));
    // convolutie en berekenen gradient
    float gx = tl + 2.0*l + bl - tr - 2.0*r - br;
    float gy = -tl - 2.0*t - tr + bl + 2.0*b + br;
    float grad = sqrt((gx*gx) + (gy*gy));
    // limiet testen
    if (grad > (ts_c*ts_c)){
        gl_FragColor.xyz = vec3(0.0,0.0,0.0);
    }
}

```

pixel een donkere kleur toegekend: de pixel is dan deel van de contourlijn. Met de GLSL-functie `texture2D` kan het domein van de depth map uniform bemonsterd worden. De GLSL-code van de fragment shader is opgenomen in codeblok 6.5.

### 6.3.3 Evaluatie

#### Performantie

In tabel 6.3 is te zien hoe het Sobel Filter-algoritme interactieve framerates biedt voor alle modellen in de benchmark. Het algoritme presteert zelfs gemiddeld 10% beter dan het Threshold-algoritme uit sectie 6.2. Dit is te wijten aan het feit dat het toepassen van de Sobel Filter zeer efficiënt kan geïmplementeerd worden in de fragment shader. Ook is de afname in complexiteit van de tussentijdse render (depth map i.p.v. diffuse rendering) niet in verhouding met de toename van het werk in de signaalanalyse: deze laatste blijft immers constant voor elk model.

Model	Gem. FPS	% t.o.v. 6.1
Triceratops	135	+9%
Cow	113	+9%
Elephant	81	+9%
Armadillo	25	+12%
Heptoroid	34	+8%

Tabel 6.3: Resultaten van benchmarks met Sobel Filter-algoritme 6.4, uitgevoerd op een resolutie van 512 x 512 pixels.

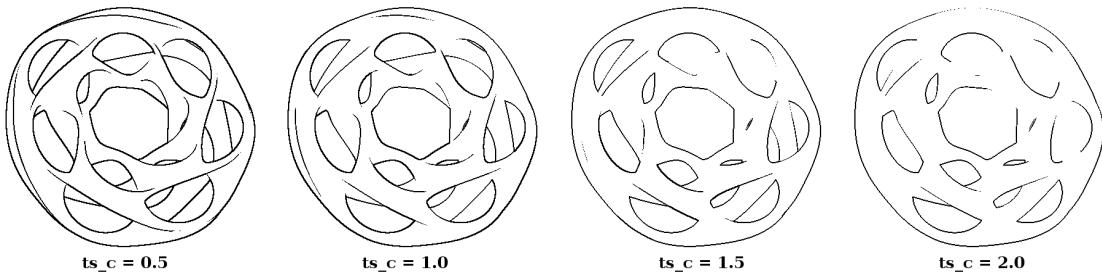
Ook werd nagegaan hoe algoritme 6.4 schaalt bij grotere resoluties. Bij Image Space algoritmes heeft de resolutie wel degelijk een invloed op de computatietijd, zoals beschreven in sectie 3.2.4. In tabel 6.4 ziet men hoe het verdubbelen van de resolutie slechts een beperkte impact heeft op de performantie van het algoritme, uitgevoerd op het *Armadillo*-testmodel. Zelfs bij erg hoge resoluties zoals 4096 x 4096 pixels, waar de depth map ongeveer 16 miljoen z-waarden bevat, blijft de framerate voldoende voor interactieve toepassingen.

Resolutie	Gem. FPS
256 x 256	25
512 x 512	23
1024 x 1024	21
2048 x 2048	20
4096 x 4096	19

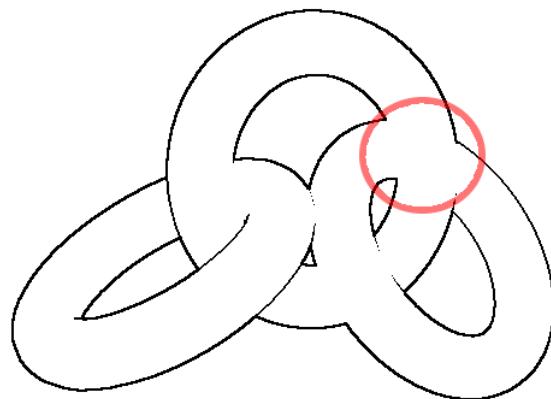
Tabel 6.4: Resultaten van benchmarks met het Sobel Filter-algoritme 6.4, uitgevoerd op het *Armadillo*-testmodel (345 944 polygonen) bij verschillende resoluties.

## Correctheid

In afbeelding 6.7 is te zien hoe het Sobel Filter-algoritme minder lijndikte-gerelateerde problemen heeft dan het Threshold-algoritme: indien de limiet verhoogd wordt, verdwijnen er gewoon meer contourlijnen, zoals verwacht. De bestaande contourlijnen worden bij het verhogen van de limiet minder uitgesproken, maar dit gebeurt op gelijke wijze voor *alle* lijnen, met als gevolg dat de totale consistentie van de figuur bewaard blijft.



Figuur 6.7: Het toepassen het Sobel Filter algoritme 6.4 op het Heptoroid-testmodel met verschillende waarden voor de limiet  $ts_c$ . Alle contourlijnen worden in gelijke mate dunner bij het verhogen van de limiet  $ts_c$ .



Figuur 6.8: Het verdwijnen van interne contourlijnen als de limietwaarde  $ts_c$  te hoog wordt ingesteld in algoritme 6.4.

### Temporele coherentie

De temporele coherentie van opeenvolgende frames gegenereerd met algoritme 6.4 is goed, maar er is wel een probleem op te merken bij zogenaamde *interne contouren*. Dit zijn contourlijnen die voorkomen *binnenin* de figuur: in de depth map zijn dit plaatsen waar de overgang niet naar  $\infty$  is, maar naar een ‘diepere’ z-waarde. Merk op dat het hier nog altijd geldige contourlijnen betreft, gedefinieerd door de Contour Generator. Op figuur 6.5 vind men bijvoorbeeld een interne contour bij de overlappende stukken tussen de twee driehoeken.

Het probleem is dat de gradiëntwaarde in deze gebieden kleiner is dan die bij een overgang naar een punt op oneindig: indien de limietwaarde  $ts_c$  te hoog is ingesteld kunnen deze interne contourlijnen in opeenvolgende frames verschijnen en verdwijnen, wat voor een storend effect zorgt. Op figuur 6.8 wordt dit effect gedemonstreerd aan de hand van een testmodel, opgesteld uit 3 ringen die in elkaar geschakeld zijn: op de plaatsen waar de ringen elkaar kruisen ontstaan normaalgezien interne contouren, maar deze worden door het Sobel Filter-algoritme weggefilterd.

### 6.3.4 Besluit

Door het implementeren van de Sobel Filtering-techniek in een fragment shader kan algoritme 6.4 aan interactieve snelheden normale contourlijnen tekenen. Hierbij wordt de controle over de lijndikte beter behouden dan met het Threshold-algoritme uit sectie 6.2. Indien men enkel contourlijnen wilt renderen is algoritme 6.4 dus de aangewezen keuze: er kunnen resultaten mee gegenereerd worden die van dezelfde kwaliteit zijn als behaald wordt met een Object Space algoritme zoals bijvoorbeeld 4.2.

## 6.4 Suggestieve contouren via radiale filter

In deze sectie wordt een door [Decarlo et al., 2003] geïntroduceerd algoritme besproken om suggestieve contourlijnen te extraheren en te tekenen in Image Space. Dit algoritme werd in het kader van deze thesis geïmplementeerd op de GPU, waardoor het voor interactieve toepassingen toepasbaar wordt.

### 6.4.1 Theorie

#### Valleidetectiemethodes

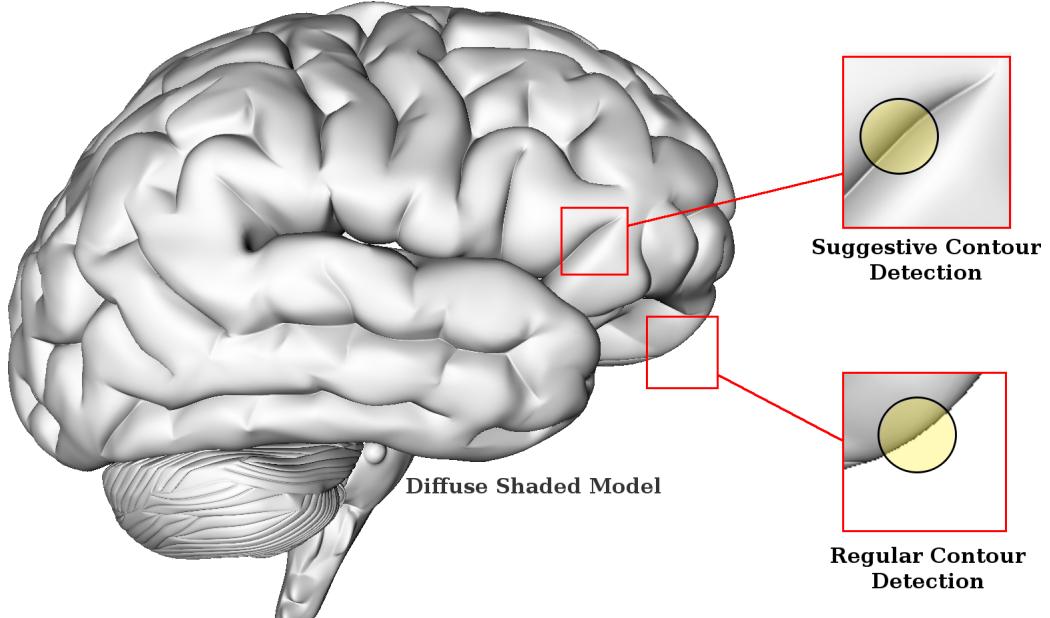
In de signaalverwerkingsstap van algoritmes 6.1 en 6.4 moesten telkens normale contourlijnen gevonden worden: plotse veranderingen in respectievelijk  $\mathbf{n} \cdot \mathbf{v}$  en de z-buffer waarde van de pixel in de tussentijdse render. Aangezien *suggestieve* contourlijnen zich per definitie in valleien bevinden op het model (zie bijvoorbeeld afbeelding 4.6), is in de signaalverwerkingsmethode hiervoor een *vallei-detector* vereist, die op basis van een bepaald scalair veld de gebieden waar er zich een vallei vormt kan extraheren.

Er zijn vele signaalverwerkingsmethodes die valleien kunnen detecteren [Lopez et al., 1999], maar deze methoden zijn meestal erg complex, en hierdoor ongeschikt voor real-time extractie. Het zijn immers algemene methodes die ontwikkeld zijn om rekening te houden met eventuele ruis in bestaande beelden. Er wordt in image processing immers vaak van een reëel beeld (foto) vertrokken, en niet van computergegenererde beelden, zoals het geval in de Image Space algoritmes die hier behandeld worden.

#### Valleidetectie op een ruisvrije afbeelding

Net omdat we zelf kunnen garanderen dat het gegenereerde scalaire veld ruisvrij is, stelt men een vereenvoudigd algoritme voor om valleien te detecteren. Dit algoritme baseert zich niet op de Suggestieve Contour Generator, maar op de eigenschappen van een vallei in functie van  $\mathbf{n} \cdot \mathbf{v}$ , en meerbepaald de kleur van de pixels in een vallei. Er wordt gewerkt op een diffuse render van het model, gegenereerd met de techniek beschreven in sectie 6.2.1.

Een pixel die in een vallei ligt zal niet noodzakelijk de minimum intensiteit hebben in zijn omgeving, maar wel behoren tot een kleine set van donkere pixels die door de



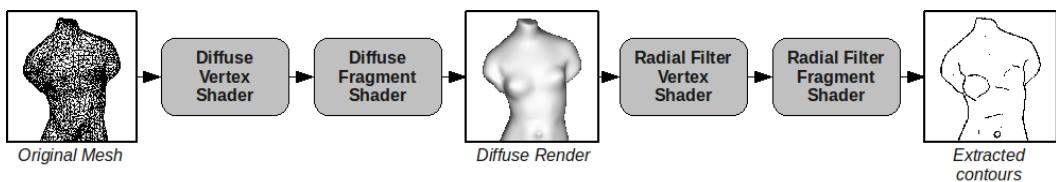
Figuur 6.9: Het radiale-filter algoritme detecteert zowel suggestieve als normale contourlijnen door pixelintensiteiten te vergelijken in een gebied rond de centrale pixel (aangeduid met een gele markering).

omgeving ‘snijden’. Als de valleiwanden steil zijn, zullen er zich in een nabij gebied ook enkele zeer heldere pixels bevinden. Men kan een bepaalde pixel  $p_i$  dus tot een vallei rekenen als:

- Slechts een bepaald percentage  $s$  van alle pixels in een radiaal gebied rond  $p_i$  heeft een intensiteit kleiner dan die van  $p_i$ .
- Het verschil in intensiteit  $p_{max} - p_i$ , waarin  $p_{max}$  de pixel heeft met de hoogste intensiteit in een radiaal gebied rond  $p_i$ , kleiner is dan een bepaalde threshold  $d$ .

Om de effecten van de discretisatie tegen te gaan wordt in [Decarlo et al., 2003] voorgesteld om het percentage  $s$  te laten variëren als  $(1 - \frac{1}{r})$  en  $d$  op gelijke wijze als  $r$ , waarin  $r$  de straal voorstelt van het gebied rond de pixel  $p_i$  waarin gezocht wordt. We benoemen  $r$  als de *radiale parameter*.

Buiten de afname in complexiteit t.o.v. traditionele valleidetectiealgoritmes is er nog een bijkomend voordeel: ook *normale contourlijnen* worden gedetecteerd met dit algoritme: deze zullen immers ook een kleine set donkere pixels vormen met in onmiddellijke nabijheid een set zeer heldere pixels (in dit geval: de witte achtergrond). Merk op dat deze operatie onafhankelijk is voor elke pixel  $p_i$  en bijgevolg in parallel kan uitgevoerd worden: uitermate geschikt voor een GPU-gebaseerde implementatie. Het radiale filter-algoritme wordt gedemonstreerd op figuur 6.9, waarop te zien is



Figuur 6.10: Overzicht van de implementatie van het radiale filter-algoritme voor extractie van contourlijnen.

hoe de intuïtieve beschrijving van ‘een donkere set pixels temidden van heldere pixels’ geldt voor zowel suggestieve als normale contourlijnen.

#### 6.4.2 Implementatie

De implementatie van het radiale filter-algoritme werd als volgt praktisch uitgewerkt in OpenGL/GLSL: per frame wordt, net zoals in algoritme 6.1, een diffuse rendering gemaakt van het model, die wordt opgeslagen in een textuur. Vervolgens wordt een vierkant getekend met als zijde de gewenste resolutie. In de vertex shader wordt de perspectief/viewtransformatie uitgevoerd en worden de textuurcoördinaten gekopieerd. De Pseudo Code voor de CPU-code en de vertex shader van de radiale filtering is dus zeer gelijkend op respectievelijk 6.1 en 6.2.

Het interessante deel van het radiale filter-algoritme is te vinden in de fragment shader. Deze implementatie werkt met een vierkant gebied rond de pixel  $p_i$ . De intensiteitsberekeningsmethode `intensity(vec4 rgb)` is dezelfde als in de Sobel Filter-shader 6.5. In codeblok 6.6 wordt de GLSL-code voor de fragment shader gegeven.

#### 6.4.3 Evaluatie

##### Performantie

Algoritme 6.5 biedt interactieve framerates voor alle modellen uit de benchmarkset, zoals te zien in tabel 6.5. Deze test werd uitgevoerd voor verschillende radiuswaarden  $r$ : indien voor deze radius een hogere waarde wordt genomen, moeten er per pixel  $p_i$  meer intensiteiten vergeleken worden van de nabije pixels om tot een beslissing te komen. Vooral voor modellen met een lagere polycount heeft deze parameter een duidelijke invloed op de framerate: bij modellen met een hogere polycount is dit effect minder uitgesproken, omdat daar de snelheid van de diffuse belichtingsstap de limiterende factor is: er moeten meer vertices verwerkt worden in de vertex shader om het diffuse lichtmodel te evalueren.

Net zoals bij de andere Image Space algoritmes heeft ook de resolutie een invloed op de performantie, zoals aangetoond in tabel 6.6. De framerate blijft echter wel voldoende voor interactieve toepassingen, zelfs bij erg hoge resoluties.

Codeblok 6.6: S. Contouren via Radiale Filter - Fragment Shader (GLSL)

```
// textuur met diffuse rendering
uniform sampler2D diffuse;
uniform int RADIUS;
uniform int WIDTH;

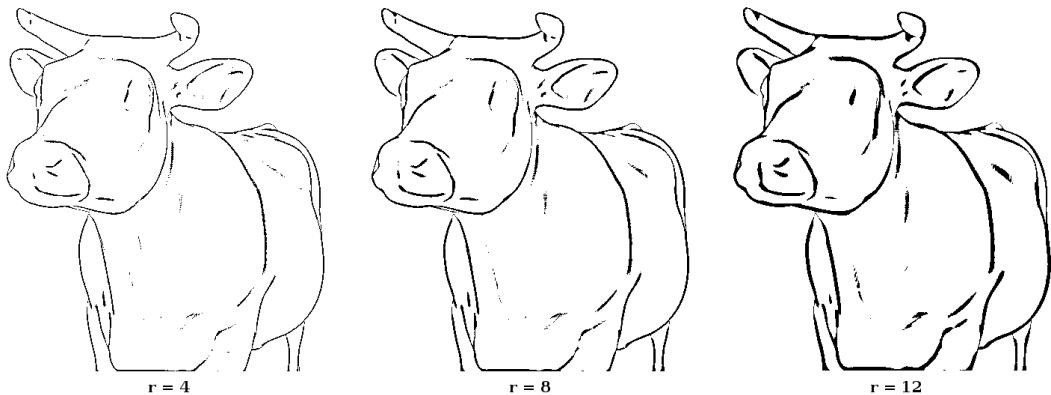
void main(void)
{
    gl_FragColor.xyz = vec3(1.0,1.0,1.0) // background
    float s = 1.0/WIDTH; // step width
    vec2 center_color = gl_TexCoord[0].st;
    // de centrale pixel p_i
    float center_intensity = intensity(texture2D(color, center));
    int darker_count = 0;
    float max_intensity = center_intensity;
    // zoek donkerdere pixels en helderste pixel
    for(int i = -RADIUS; i <= RADIUS; i++){
        for(int j = -RADIUS; j <= RADIUS; j++){
            float current_intensity = intensity(texture2D(diffuse,
                center+vec2(i*s,j*s)));
            if(current_intensity < center_intensity){
                darker_count++;
            }
            if(current_intensity > max_intensity){
                max_intensity = current_intensity;
            }
        }
    }
    // is p_i een valleipixel?
    if((max_intensity - center_intensity) > 0.01*radius){
        if(darker_count/(radius*radius) < (1-(1/radius))){
            gl_FragColor.xyz = vec3(0.0,0.0,0.0); // ja
        }
    }
}
```

Model	$r = 2$	$r = 3$	$r = 5$	$r = 10$
Triceratops	81	75	61	45
Cow	65	61	56	40
Elephant	50	45	43	36
Armadillo	22	21	20	18
Heptoroid	29	28	26	22

Tabel 6.5: Gemiddelde framerate (in frames/seconden) bij het uitvoeren van de benchmarks met 6.6 voor het extraheren van suggestieve en normale contourlijnen, met verschillende waarden voor de radiale parameter  $r$ , die het gebied rond de pixel  $p_i$  bepaalt waarin gezocht wordt naar pixelintensiteiten.

Resolutie	Gem. FPS
$256 \times 256$	63
$512 \times 512$	61
$1024 \times 1024$	51
$2048 \times 2048$	45
$4096 \times 4096$	41

Tabel 6.6: Resultaten van benchmarks met het radiale filter-algoritme 6.6, uitgevoerd met radiale parameter  $r = 5$  op het Cow-testmodel (92 864 polygonen), bij verschillende resoluties.



Figuur 6.11: Het Cow-testmodel gerenderd met het radiale filter-algoritme 6.6, met verschillende waarden voor de radiale parameter  $r$ .

### Correctheid

Op figuur 6.11 wordt het Cow-testmodel gerenderd met verschillende waarden voor de radiale parameter  $r$ . Men kan opmerken dat het variëren van de radiale parameter logischerwijs de lijndikte beïnvloedt, en dit op enkele uitzonderingen na voor alle lijnen op dezelfde manier gebeurt. Wel zijn er (vooral op de figuur gegenereerd met  $r = 4$ ) lichte aliasing-effecten te bemerken op de gegenereerde contourlijnen: deze kunnen, zoals [Decarlo et al., 2003] aangeeft, verholpen worden met een *median filter* of met een eenvoudige *blur* filter die werkt met dezelfde straal  $r$ . Dit kan echter niet in dezelfde *pass* als de radiale filter, gezien eerst voor alle naburige pixels de kleur bepaald moet zijn.

Uiteraard is de kwaliteit van de afbeeldingen gegenereerd met het radiale filter-algoritme voor contourlijnextractie ook sterk afhankelijk van de kwaliteit van de diffuse shading, zoals al werd aangegeven in sectie 6.2.3.



*Figuur 6.12: Probleem bij het toepassen van het radiale filter-algoritme waarbij  $r$  afhankelijk is van de afstand tot het camerastandpunt  $\mathbf{c}$ : de contourlijnen op het hoofd van het Triceratops-testmodel zijn dikker dan de contourlijnen op de rest van het model.*

### Temporele coherentie

Indien er wordt ingezoomd op het model, krimpen de aanwezige contourlijnen. Dit is als volgt verklaarbaar: Als positie van de lichtbron in het diffuse belichtingsschema werd de positie  $\mathbf{c}$  van de camera genomen. Als de camera dichter bij de figuur wordt gezet zal de diffuse rendering dus veranderen, wat het radiale filter-algoritme beïnvloedt. Doch, zelfs als de lichtbron op een vast punt zou gezet worden blijft dit probleem bestaan: de valleien bestaan uit bredere stroken zwarte pixels, waardoor algoritme 6.6 kan falen als er zich *alleen* valleipixels in het gebied behandeld door de radiale filter met straal  $r$  bevinden. De maximumintensiteit in het gebied verschilt onvoldoende van de intensiteit van de behandelde pixel  $p_i$ , waardoor de test  $(p_{max} - p_i)$  faalt.

Dit probleem kan opgelost worden door niet alleen met de renderbreedte rekening te houden bij de radiale filtering, maar ook met de relatieve afstand van de camera tot het object: hoe dichter het camerastandpunt  $\mathbf{c}$  zich bij het object bevindt, hoe groter de radiale parameter  $r$  moet zijn om de contourlijnen te kunnen extraheren. Deze methode heeft dan weer als nadeel dat camerastandpunten waarin zowel geometrie die dichtbij is als geometrie die relatief verder ligt te zien zijn aanleiding geven tot beelden waarin de contourlijnen op de verschillende niveaus duidelijk een verschillende lijndikte hebben. Een voorbeeld wordt gegeven in afbeelding 6.12.

#### 6.4.4 Besluit

Ondanks de kleine temporele coherentie-problemen die de vereenvoudiging van traditionele complexe vallei-algoritmes met zich meebrengt is de GPU-implementatie van het radiale filter-algoritme 6.6 een performante en verkiesbare manier om suggestieve contourlijnen te tekenen in Image Space. Door het efficiënt implementeren van de radiale filter in een fragment shader maakt het algoritme optimaal gebruik van de functionaliteiten in hedendaagse grafische hardware.

## Hoofdstuk 7

# Resultaten en Besluit

In deze masterproef werden verschillende algoritmes voor het berekenen en tekenen van (suggestieve) contourlijnen opgesteld, geïmplementeerd en vergeleken. In dit besluit bespreken we de eigen inbreng, resultaten, het geleverde werk en de mogelijkheden tot verder onderzoek.

### 7.1 Eigen inbreng

In deze masterproef werden de volgende zaken gerealiseerd:

- **Vergelijking en implementatie:** De bestaande algoritmes voor het berekenen en tekenen van normale en suggestieve contourlijnen werden efficiënt geïmplementeerd en vergeleken.
- **Performantieverbeteringen:** Voor de bestaande Object Space-algoritmes werden performantieverbeteringen doorgevoerd en geïmplementeerd in sectie 4.4. Ook werden nieuwe parameters ingevoerd om o.a. de temporele coherentie te verbeteren in sectie 4.5.3 (de *discovery parameter*  $\delta$ ).
- **GPU-shader algoritme:** In sectie 5.4 werd een nieuw algoritme voorgesteld dat de Object Space-aanpak toepast op de GPU. Door het gebruik van vertex/fragment shaders en het introduceren van de nieuwe limieten  $t_c$  en  $t_{sc}$  werd een uiterst performant algoritme geïmplementeerd, dat niet hoeft in te boeten aan kwaliteit tegenover de traditionele CPU-gebaseerde methodes uit hoofdstuk 4.
- **GPU-implementatie Image Space algoritmes:** In hoofdstuk 6 werden de bestaande Image Space algoritmes voor het eerst geïmplementeerd op de GPU, waardoor ze ook voor interactieve toepassingen kunnen gebruikt worden.

## 7.2 CPU-gebaseerde algoritmes

In hoofdstuk 4 over CPU Object Space algoritmes werd het basisalgoritme voor suggestieve contourlijnen uit [Decarlo et al., 2003] geïmplementeerd. Na het implementeren van enkele temporele coherentie-verbeteringen, voorgesteld in sectie 4.6, werden met dit algoritme mooie resultaten geboekt: zie figuur 7.1 voor enkele voorbeelden.

### CPU-gebaseerde Object Space algoritmes: mathematisch correct

De ontwikkelde CPU-gebaseerde Object Space algoritmes zijn mathematisch correct: ze baseren zich rechtstreeks op de definitie van de Contour Generator (2.1) en de Suggestieve Contour Generator (2.3). Alle contourlijnsegmenten worden gevonden: een subset daarvan tekenen we ter bevordering van de temporele coherentie niet, maar hun geometrische locatie is wél bepaald en beschikbaar voor verdere analyse. Dit zorgt ervoor dat we de resultaten bij het gebruik van een CPU-gebaseerd Object Space algoritme als een referentie kunnen gebruiken voor andere methodes om contourlijnen te extraheren en te tekenen.

Waar we de CPU-gebaseerde Object Space algoritmes echter moeilijker voor kunnen gebruiken is de weergave van objecten in real-time toepassingen. Voor het basisalgoritme 4.2 voor het tekenen van normale contourlijnen neemt de performantie exponentieel af met de complexiteit van het gebruikte 3D-model. Dit werd verbeterd in algoritme 4.2, door de normale contourlijnen niet *over* de polygonen te tekenen, maar op de scheidingslijnen tussen de polygonen. Deze vereenvoudiging zorgt echter wel voor storende visuele problemen bij modellen met een lage polygon count.

Voor suggestieve contouren bleek dat normale contouren en suggestieve contouren samen tekenen met het basisalgoritme 4.3 performant genoeg was voor eenvoudige 3D-modellen, maar een onaanvaardbare performantie had (gemiddeld  $< 3$  frames/ seconde) voor complexere modellen. In sectie 4.4 werden verbeteringen voorgesteld die de performantie van het algoritme voor grote modellen ongeveer verdubbelen. Hoewel dit op zich een verdienstelijke bijdrage is, blijft het belangrijk om het fundamentele probleem van Object Space algoritmes te erkennen.

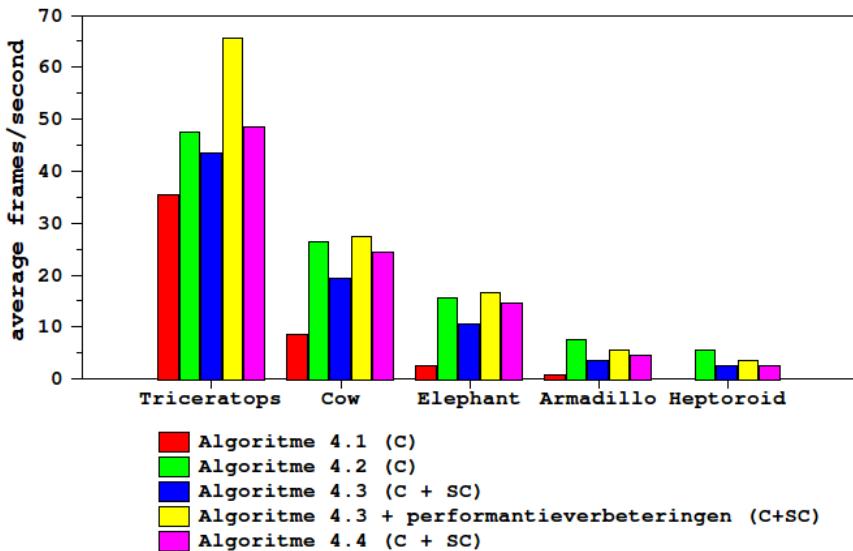
### Performantie: een fundamenteel probleem

De fundamentele oorzaak van deze performantie-afname is de volgende: aangezien we met Object Space-gebaseerde algoritmes mathematisch gezien alle contourlijnsegmenten willen vinden, zijn computationeel zware berekeningen nodig om de waarden voor elke vertex van het gediscretiseerde model te kennen, alsook efficiënte datastructuren om deze waarden te organiseren. De grootte van deze datastructuren groeit minstens lineair met de complexiteit van het model, en (afhankelijk van de toegepaste verbeteringen) zelfs exponentieel. Simpelweg uitgedrukt: meer vertices, meer werk.

De performantieverbeteringen die werden voorgesteld in sectie 4.4 zorgen er wél voor dat er *minder tests* moeten worden uitgevoerd op *minder vlakken*, maar op deze manier kan men maar zo ver optimaliseren tot men (hypothetisch gesteld) enkel nog



Figuur 7.1: Resultaten bereikt met de Object Space algoritmes uit hoofdstuk 4.



Figuur 7.2: Performantieresultaten voor alle algoritmes uit hoofdstuk 4. In de grafiek wordt aangegeven of er normale contouren (C) en/of suggestieve contouren (SC) werden gerenderd. Hoewel zowel de performantieverbetering voor algoritme 4.3 als het stochastische algoritme 4.4 de performantie verbeteren, blijft de framerate voor complexe modellen te laag om in interactieve toepassingen aangenaam te werken.

de vlakken bezocht waar er daadwerkelijk een contourlijn aanwezig is: zelfs dan is het lokaliseren van het contourlijnsegment op het vlak geen triviale opdracht en blijft er een vaste kost voor de rekentijd.

In sectie 4.5 werd een performantieverbetering voorgesteld die de eis van het vinden van *alle* contourlijnsegmenten afzwakte. Door het kiezen van een willekeurige subset van de vlakken en het volgen van de contourlijnsegmenten over het oppervlak van het gediscretiseerde model kan men na enkele frames een voldoende aantal contourlijnen vinden. Het bijhouden van deze structuren en het actief volgen van de contourlijnsegmenten vereist echter ook een efficiënte implementatie. Dit probleem zorgt ervoor dat ook de performantieresultaten van de algoritmes gebaseerd op deze techniek weinig verbetering brengen, maar een interessant uitgangspunt vormen voor verder onderzoek.

Een overzicht van de performantie van alle algoritmes uit hoofdstuk 4 wordt gegeven in grafiek 7.2.

### Toepassingen van CPU-gebaseerde Object Space algoritmes

De algoritmes ontwikkeld in hoofdstuk 4 kunnen ondanks de performantieproblemen toch nuttig worden toegepast: zoals eerder aangehaald kunnen ze gebruikt worden om een referentiebeeld te vormen om de correctheid van andere contourlijn-algoritmes mee te vergelijken.

Ook valt op in grafiek 7.2 dat de framerate voor minder complexe objecten op

een aanvaardbaar niveau kan gebracht worden met de voorgestelde verbeteringen. Hierdoor kunnen deze gebruikt worden voor interactieve toepassingen, bijvoorbeeld op systemen waar beperkte of geen grafische hardware aanwezig is (mobiele toestellen, low-end spelconsoles, ...). Om de complexere objecten aan een interactieve snelheid weer te geven kunnen hun 3D-meshes ook vereenvoudigd worden door middel van *smoothing*: hierbij gaat wel detail over de curvatuur verloren, gezien er vertices van het model verwijderd worden.

### Verder onderzoek

Verder onderzoek binnen dit topic zou zich kunnen concentreren op efficiëntere manieren om met grote hoeveelheden vertex data in het systeemgeheugen te werken. Ook kan er nog veel gebeuren op het gebied van *dynamic level-of-detail*, een eigenschap die aan Object Space algoritmes ontbreekt, zoals beschreven in sectie 3.2.4. Men zou bijvoorbeeld dynamisch (afhankelijk van het camerastandpunt) delen van de mesh kunnen *smoothen* om enkel detail weer te geven in de delen van het object waarop de focus momenteel ligt. Ook zouden bij het dichterbij/verderaf brengen van het camerastandpunt meer of minder contourlijnen kunnen getekend worden.

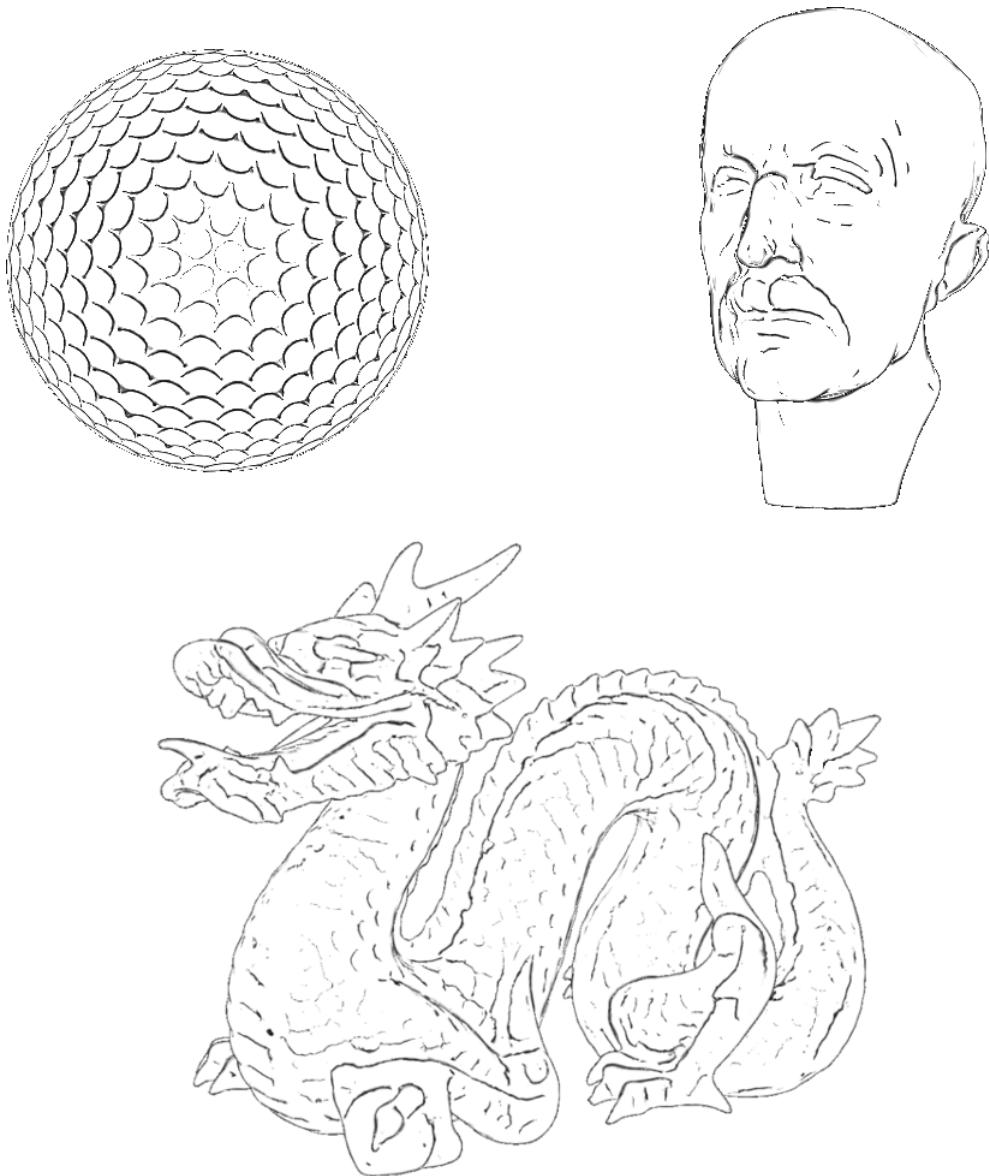
## 7.3 GPU-gebaseerde algoritmes

Uit de analyses in hoofdstuk 4 en de performantietests in grafiek 7.2 werd onder meer duidelijk dat we voor het behalen van interactieve framerates voor complexe modellen gebruik moeten maken van de functionaliteit van de GPU. We willen echter dat de correctheid van de oplossing daar niet of zo min mogelijk onder lijdt. Daarom werden eerst Object Space algoritmes onderzocht: deze blijven dicht bij de originele definities van de Contour Generatoren.

### Een sterk verbeterd Object Space GPU algoritme

De problemen van het Texture Mapping-algoritme 5.1, een reeds bestaand GPU-versneld algoritme voorgesteld in [Decarlo et al., 2004], werden in sectie 5.3 blootgelegd. De performantie van het algoritme blijft ondermaats omdat voor de computationeel zware berekening van  $\kappa_r$  en  $D_w\kappa_r$  geen gebruik wordt gemaakt van de GPU. Het verplaatsen van de limiet-tests die beslissen of een punt al dan niet tot de (suggestieve) contourlijn behoort naar het texture-mapping proces brengt ook verschillende problemen met zich mee: texture lookups zijn traag, en bovendien zorgt de interpolatie voor een inconsistente lijndikte en storende artefacten.

Deze problemen werden aangepakt door een nieuw GPU-shader algoritme 5.2 voor te stellen. Door het invoeren van de nieuwe limieten  $t_c$  en  $t_{sc}$ , die de lokale curvatuur van het object in rekening brengen, en het uitvoeren van de berekening van  $\kappa_r$  en  $D_w\kappa_r$  in een vertex shader konden zowel de correctheid als de performantie op een hoger niveau getild worden. Op figuur 7.3 worden enkele resultaten gegeven die met algoritme 5.2 bereikt werden.

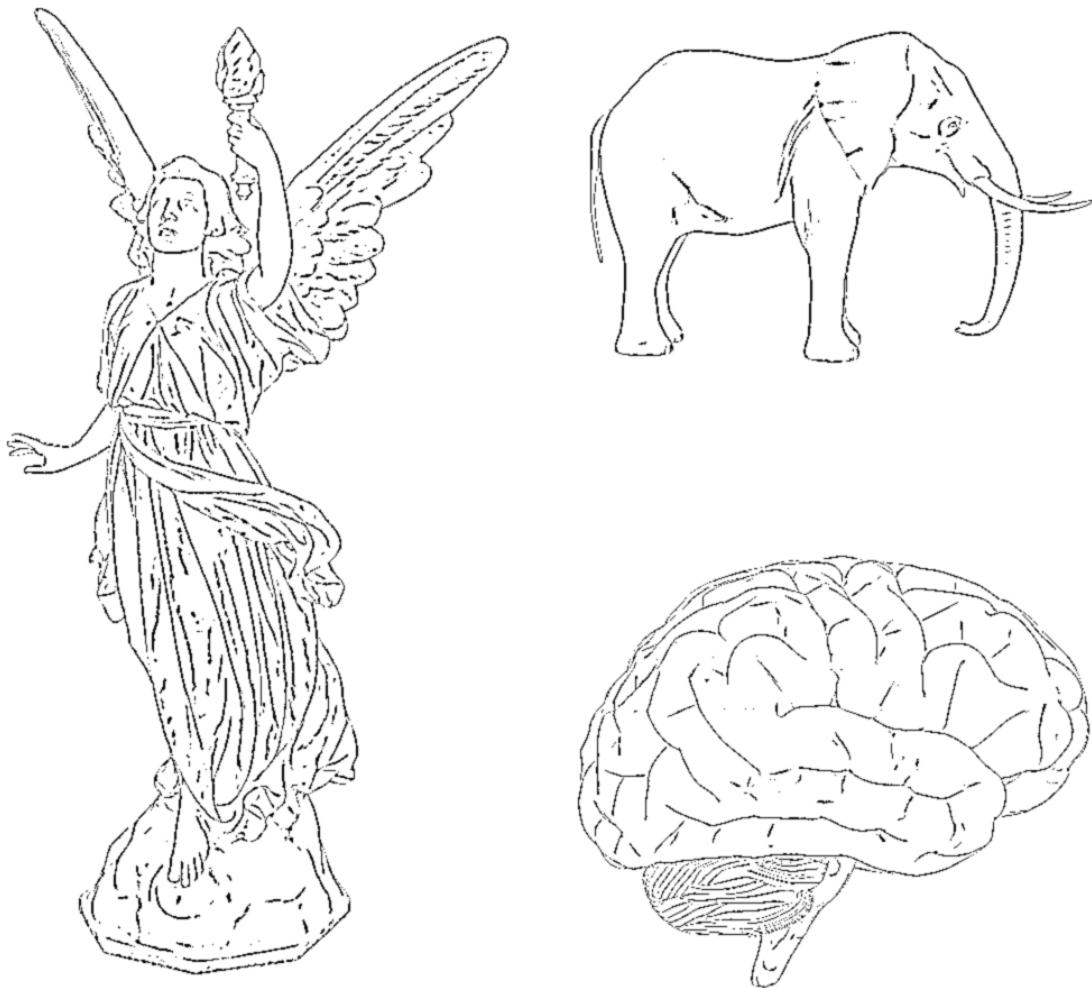


Figuur 7.3: Resultaten bereikt met het Object Space GPU algoritme 5.2.

De gegenereerde lijnsegmenten hebben niet overal *exact* dezelfde lijndikte, maar de resultaten zijn veel beter dan die bereikt met de textuurinterpolatie in algoritme 5.1. De segmenten werden ook voorzien van een standaard fading-schema, zodat de temporele coherentie aanvaardbaar is.

#### Het behouden van lijnstijl-controle in Image Space algoritmes

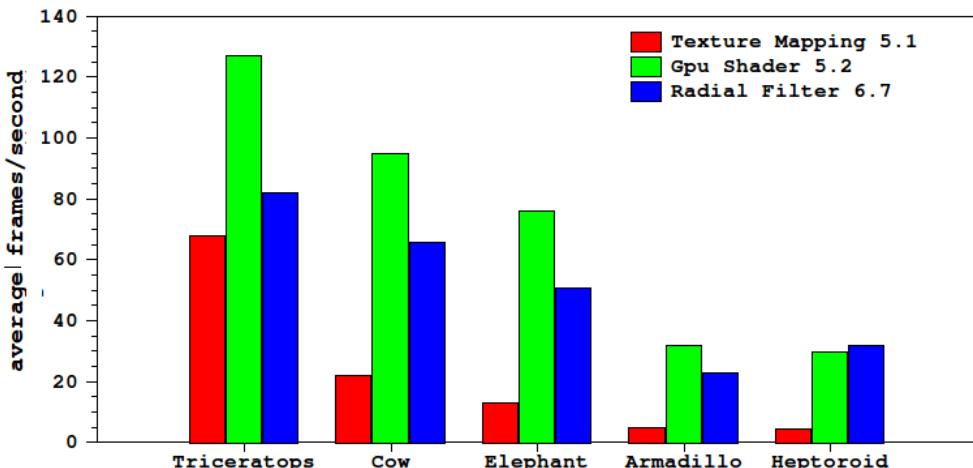
In hoofdstuk 6 werden verschillende algoritmes opgesteld en getest om contourlijnen en suggestieve contourlijnen via Image Space methodes te tekenen. Het grootste



Figuur 7.4: Resultaten bereikt met het Image Space Radiale Filter-algoritme 6.6.

probleem situeerde zich zoals te verwachten steeds rond de controle over de lijndikte in het uiteindelijke resultaat. Dit is een inherent probleem van Image Space methodes.

Door het balanceren van het werk tussen de tussentijdse renderingstap en de signaalverwerkingsmethode werd het Radiale Filter-algoritme 6.6 voorgesteld: hierin wordt door gebruik te maken van de eigenschappen van de tussentijdse diffuse rendering een eenvoudige valleidetectiemethode opgesteld die zowel normale als suggestieve contourlijnen kan extraheren. Door dit algoritme te implementeren op de GPU door middel van shaders kan dit alles gebeuren aan interactieve framerates, én met redelijk behoud van de controle over de lijnstijl. Enkele resultaten worden weergegeven in figuur 7.4.



Figuur 7.5: Performantieresultaten van de algoritmes uit hoofdstuk 5 en 6, bij het tekenen van normale contouren en suggestieve contouren in dezelfde pass.

### Performantieboost via de GPU

De performantie van de GPU-gebaseerde algoritmes is zowel in de Object Space als in de Image Space-aanpak erg goed. De performantieresultaten van de verschillende algoritmes uit hoofdstuk 5 en 6 worden vergeleken in grafiek 7.5. Erg opvallend is de verbetering die het GPU-shader algoritme 5.2 biedt tegenover het Texture Mapping algoritme 5.1, dat tot voor kort de enige besproken methode was die in Object Space opereerde én gebruik maakte van GPU-functionaliteit.

Voor de Image Space algoritmes is de computatietijd van signaalverwerking die gebeurt op het gerenderde scalaire veld voor alle modellen exact dezelfde. Het performantieverzil bij complexere modellen dat toch nog duidelijk aanwezig is wordt enkel en alleen bijgedragen door de tussentijdse renderingstap: voor een complexer model duurt het langer om de diffuse belichting uit te rekenen, ook al is diffuse belichting een zeer eenvoudig lichtmodel.

Bij Image Space algoritmes speelt de resolutie van de uitvoer ook een rol: signaalverwerking op een groter scalair veld vereist meer tijd. In hoofdstuk 6 werd echter ondervonden dat resolutie momenteel niet de grootste bottleneck vormt voor contourlijnextractie-methodes: de framerates namen slechts lineair af bij een verdubbeling van de resolutie.

### Verder onderzoek

Hoewel de GPU-gebaseerde algoritmes de performantie al sterk verbeteren tegenover hun CPU-gebaseerde tegenhangers, is er nog veel ruimte voor verbetering: door gebruik te maken van de moderne GPGPU (*General Purpose GPU*)-technieken moet bijvoorbeeld het absolute verschil tussen een vertex shader en een fragment shader niet meer gemaakt worden: vanuit dat vertrekpunt kunnen een plethora aan

optimalisaties worden doorgevoerd.

Een andere mogelijkheid tot verbetering ligt in de configuratie van de renderingparameters. Neem GPU-shader algoritme 5.2 als voorbeeld: hoewel  $t_c$  en  $t_{sc}$  al onafhankelijk gemaakt werden van de grootte van het gebruikte object (via de *feature size*) is voor het krijgen van een consistente figuur toch nog manuele fine-tuning van de limieten nodig. Verdere verbanden hiertussen zouden kunnen gezocht en gebruikt worden om nieuwe tests te introduceren.

Bij Image Space algoritmes kan ook gewerkt worden aan zogenaamde *post-processing*. De gegenereerde beelden bevatten vaak nog veel aliasing effecten, die kunnen weggewerkt worden met een eenvoudige blur filter. Dit is echter niet optimaal: zowel aan de oorzaak van de aliasing (het algoritme 6.6 zelf) als het behandelen van de gevolgen kan nog gewerkt worden.

## **Bijlage A**

## **Poster**

Deze bijlage bevat een poster waarop het doel van deze masterproef wordt geformuleerd en enkele behaalde resultaten worden getoond. Deze poster werd gemaakt in opdracht van het Departement Computerwetenschappen. Deze versie is een geschaalde weergave van de originele poster in A2-formaat.



KATHOLIEKE UNIVERSITEIT  
LEUVEN

FACULTEIT  
INGENIEURWETENSCHAPPEN  
1423

Master  
Computer-  
wetenschappen

Masterproef  
Jeroen Baert

Promotor  
Prof. Dr. Ir. P.  
Dutré

Academiejaar  
2009-2010

# Contourlijnen in interactieve 3D-toepassingen

Situering	Doelstelling	Resultaten
<ul style="list-style-type: none"><li>Lijntekeningen zijn opgebouwd uit <b>contourlijnen</b></li><li>Geven informatie over vorm en curvatuur.</li><li>Welke lijnen wél / niet tekenen? -&gt; <b>Niet triviaal</b>.</li><li>Lijnen mathematisch definiëren a.d.h.v differentiële geometrie.</li></ul>	<ul style="list-style-type: none"><li>Van <b>3d model naar correcte lijntekening</b>.</li><li>Algoritmes implementeren, evalueren en vergelijken.</li><li>Nieuwe technieken voorstellen en testen.</li><li>Focus op <b>performantie</b>: realtime manipulatie mogelijk maken</li></ul>	<ul style="list-style-type: none"><li>Nieuw en performant Object Space GPU-versneld algoritme.</li><li>Temporele Coherentieverbeteringen voor CPU-gebaseerde algoritmes.</li><li>Efficiënte GPU-implementatie van bestaande Image Space algoritmes.</li><li><b>Toepassingen</b>: medical imaging, computer games, art, ...</li></ul> <p>GPU-gebaseerde methodes</p> <p>CPU-gebaseerde methodes</p> <p>Suggestieve contouren</p>

Frank Miller – stijl: Suggestieve highlights

## **Bijlage B**

### **IEEE-stijl artikel**

Deze bijlage bevat een IEEE-stijl artikel waarin de belangrijkste onderdelen en conclusies van deze thesis worden toegelicht. Dit artikel werd gemaakt in opdracht van het Departement Computerwetenschappen.

Het artikel werd opgesteld in het Engels, volgens de templates en voorschriften zoals die kunnen gevonden worden op [http://www.ieee.org/publications\\_standards/publications/authors/authors\\_journals.html](http://www.ieee.org/publications_standards/publications/authors/authors_journals.html).

# Contour line extraction and rendering for interactive 3D applications

Jeroen Baert

**Abstract**— In line drawings artists/designers use different kinds of contour lines to define very different properties of the portrayed object. We want to define these contour lines mathematically, compute them and render them with computer graphics techniques.

Our goal is to succeed at rendering complex models in the line drawing style at interactive rates. This is a form of Non-Photorealistic Rendering, shorthand NPR. In this document, we present improvements to existing Object Space and Image Space algorithms and a new high performance Object Space GPU-accelerated algorithm to render these sets of contour lines.

**Index Terms**—Contours, Suggestive Contours, NPR, Rendering, 3D, Interactive Applications

## I. INTRODUCTION

We define line drawings as a visual style in which a certain object gets portrayed by using lines exclusively. The basic elements of these line drawings are the subject of this document: contour lines.

Contours are part of the most basic set of elements that can be used to convey information about an object. This visual simplicity however, does not result in an easy definition, computation or rendering of these lines.

How artists/designers select which lines to include in a certain drawing is often subject to their own interpretation and aesthetic style. Nevertheless, we can define two kinds of contour lines: *regular contour lines* and *suggestive contour lines*.

## II. CONTOUR LINES: DEFINITIONS

The definition of contour lines and suggestive contour lines is based on [1], the seminal paper when it comes to line drawings.

### A. Regular Contour Lines: Efficient Visual Cue

Informally, we can define regular contour lines as lines drawn on locations where the surface of the portrayed object turns away from the viewer and thus becomes invisible.

These lines form an important visual clue for the human object recognition system [8]. Theories in the field of neuropsychology claim that when the brain sees a contour, it is matched internally with a database of seen contours to identify the object [2]. The differentiate the object from the background, so it can be perceived as an individual element.

To define these regular contour lines mathematically, we use a structure called the Regular Contour Generator. This

describes the set of points on an arbitrary surface that, when viewed from a certain camera viewpoint, are part of a contour. This is demonstrated in image 1. This generator can be defined with the following formula:

$$\mathbf{n}(\mathbf{p}) \cdot \mathbf{v}(\mathbf{p}) = 0$$

in which  $\mathbf{p}$  is the position of a point on the surface,  $\mathbf{n}(\mathbf{p})$  the normalized normal vector in  $\mathbf{p}$ , and  $\mathbf{v}$  the view vector, which is computed using the difference between the camera position  $\mathbf{c}$  and the point  $\mathbf{p}$ .

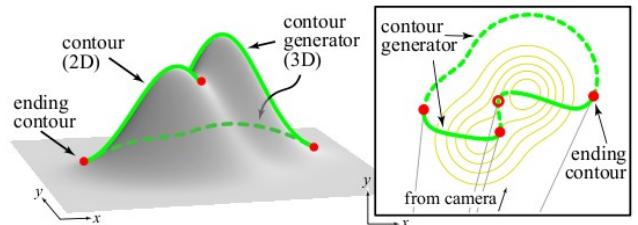


Illustration 1: The Contour Generator (image from [1])

### B. Suggestive Contour Lines: Natural addition

Often using regular contour lines only does not suffice to convey a clear image of the object. This is why an additional set of lines is needed: *suggestive contours*. These can be described informally as *almost-regular contours*: they appear in valleys where in a nearby camera viewpoint a regular contour will appear. This is how suggestive contours form a natural addition to regular contour lines, and both contour lines will smoothly fade into each other when an object is rotated.

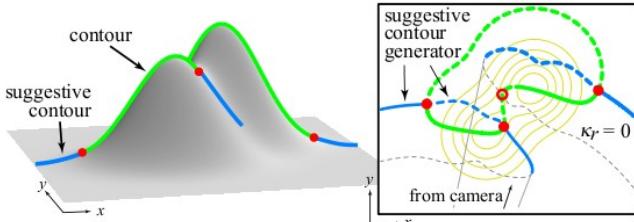
To define suggestive contours, we need the so-called radial curvature  $\kappa_r$ , which is defined as the inverse of the radius of the best approximating circle to the curve defined at a point  $\mathbf{p}$ .

$$\kappa_C(\mathbf{p}) = \frac{1}{r}$$

A first definition of the Suggestive Contour generator can be given as the locations where the radial curvature is zero and the directional derivative of the radial curvature (with a projected view vector as direction) is positive: we only want places where the surface bends away (convex) from the camera.

$$\kappa_r = 0 \text{ en } D_{\mathbf{w}} \kappa_r > 0$$

The suggestive contour generator is illustrated in the following image:



*Illustration 2: The Suggestive Contour Generator (image from [1])*

The definitions given in this section will form the basis for several algorithms which will be presented and evaluated in this work. Several alternative definitions are given in [1], but this first definition is the most important one for the algorithms we describe.

### III. OBJECT SPACE – CPU ALGORITHMS

#### A. Object Space Algorithms

We classify algorithms as *Object Space* algorithms, when they work on the original mesh data of the object: the contour lines are calculated in the original mesh coordinates, and a list of contour line segments is available after execution of the algorithm.

This has several advantages: we can style the separate lines and perform statistic analysis on their position, length and interconnectivity. The disadvantage is that these methods are often more complicated than their alternatives (Image Space methods, see further), and thus are generally slow.

#### B. Traditional CPU algorithms

First, we describe CPU-based algorithms. These algorithms don't use any kind of GPU hardware acceleration.

Several algorithms for drawing regular contour lines on a discrete mesh already exist. The most popular and straightforward ones are:

- *NdotV algorithm*: by visiting all the faces, we search for zero crossings in the  $\mathbf{n} \cdot \mathbf{v}$  dot product. When a face with a zero crossing has been found, a line segment is drawn across the face using linear interpolation. [1]
- *Appel's algorithm*: this is a simplification of the previous algorithm: instead of drawing segment lines on the faces itself, we just draw lines between front and back-facing polygons. [3]

As expected, the second algorithm is the fastest, but introduces small errors, like loops in the calculated contour lines. This effect is only visible on low-poly meshes.

To draw *suggestive* contour lines on a mesh, a similar approach as the *NdotV* algorithm is used. For this to work, we need to be able to evaluate the radial curvature  $K_r$  and its directional derivative at every vertex point. In [4], we find several formulas to do this.

For the radial curvature, we can use the Euler formula [9]:

$$\kappa_r(\mathbf{p}) = \kappa_1(\mathbf{p}) \cos^2 \phi + \kappa_2(\mathbf{p}) \sin^2 \phi$$

Where  $\kappa_1$  and  $\kappa_2$  are the principal curvatures of the surface in the vector point  $\mathbf{p}$ .  $\phi$  is the angle between the projected camera vector and the first principal direction.

To compute the directional derivative of the radial curvature in the direction of  $\mathbf{w}$  (the projected view vector  $\mathbf{v}$  on the radial plane in  $\mathbf{p}$ ), we can use the following formula (full derivation in [4]):

$$\frac{D_{\mathbf{e}_1} \kappa_1 u^3 + 3D_{\mathbf{e}_2} \kappa_1 u^2 v + 3D_{\mathbf{e}_1} \kappa_2 u v^2 + D_{\mathbf{e}_2} \kappa_2 v^3}{\|\mathbf{w}\|^2} + 2\kappa_1 \kappa_2 \|\mathbf{w}\| (1 - (\frac{\mathbf{n} \cdot \mathbf{v}}{\|\mathbf{v}\|})^2)^{-1}$$

The algorithm then searches every face of the model for faces with a zero crossing of radial curvature. Dependent on the directional derivatives, this face can contain none, one or two suggestive contour line segments.

The main problem with the traditional CPU algorithms is that they need to visit *every* face in the model and perform computationally expensive tests on them to find valid contour line segments.

This is why we propose some performance and temporal coherence improvements, based on [5]. All these improvements are based on the same principle: **we want to visit less faces, and perform less tests on them**. We still want to find *all* contours.

#### C. Performance Improvements

- *Gaussian Curvature*: We can filter faces with a positive Gaussian curvature from the model. These models have  $K = \kappa_1 \kappa_2 > 0$ , so they can contain no zero crossings of the radial curvature, and thus no suggestive contours. This filtering can be performed once as a pre-pass.

This results in filtering away an average 70% of faces, depending on the model, of course. Regular contours however, *can* occur on faces with positive Gaussian curvatures, so the filtered faces can only be used for the suggestive contours pass.

- *Backface Culling*: we don't have to compute and draw suggestive contours which are facing away from the camera. For suggestive contours, we proved that placing this test behind the test for a zero crossing of radial curvature results in faster execution time: this is due to the pre-pass filtering described above.
- *Face Normals cache*: The face normals are used to test for visibility. They can be computed in a single pre-pass and used later on.

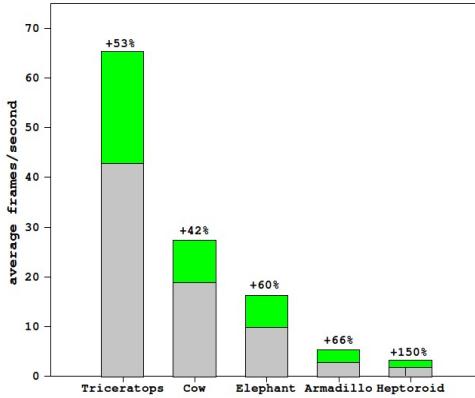
#### D. Temporal Coherence Improvements

- *Fading scheme*: By using a fading scheme, we can get rid of disturbing 'flickering' micro contours in model animations: contour lines which only just satisfy the requirements for being valid are drawn with a high alpha value.
- *Threshold on camera viewpoint*: By adding an additional limit on the camera viewpoint, we filter contours which appear 'head-on': these are too unstable to use in animations.

## E. Results

The described improvements result in a 30-50% performance boost when rendering contours and suggestive contours, as can be seen on the following graph.

For complex models, an even higher performance increase is established, but the overall frame rate is still too low for efficient interactive manipulation of the benchmark objects.



## Stochastic algorithms: an alternative

When we relax the requirement of finding *all* contours, we can apply a different approach: by selecting random faces on the model and following found contour / suggestive contour lines until we reach an end or visit a face we already visited, we can draw a high percentage of contour lines by only testing a subsection of faces. [6]

When using this technique in animation, we can retain the set of faces in which we found contour lines in the previous frame, and use them as seeds.

By adding a *discovery parameter*, we can relate the size of this seed subset to the rotation speed in animations, where seeds get invalidated faster at higher speeds. This greatly improves the percentage of contours found, as can be seen in the following graph.

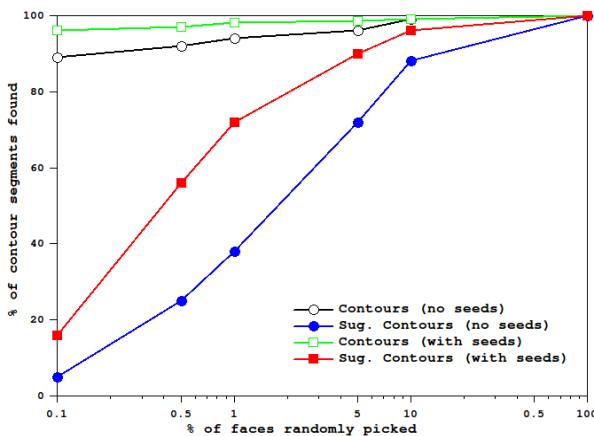


Illustration 3: Percentage of (suggestive) contour lines found when using different percentages of the total set of faces.

Unfortunately, managing the seeds introduces a new overhead, which hampers the performance boost of relaxing the requirement of finding all the seeds.

## F. Suggestive Highlights

The dual lines of suggestive contours - the parts of the loops in radial curvature where the derivative is negative – also have interesting properties: they highlight ridges of the model. Once an implementation is made for suggestive contours, just switching the signs allows the user to draw suggestive highlights too.

In combination with cartoon-style shading, this allows us to draw models in comic book-style, resembling Frank Miller's work on *Sin City*. This is demonstrated in the following figure:

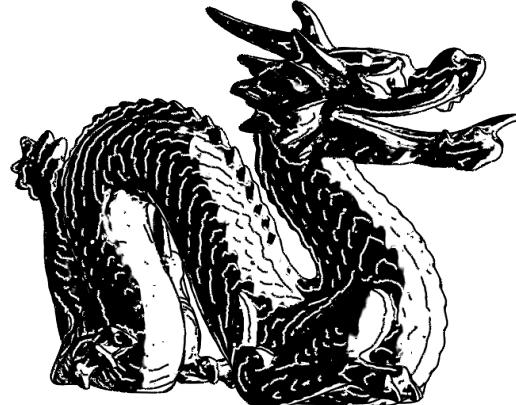


Illustration 4: Drawing suggestive highlights in combination with cartoon shading.

## IV. OBJECT SPACE – GPU ALGORITHMS

When evaluating and improving the CPU-based algorithms for contour line extraction and rendering, it became clear that if we could speed up the vertex/matrix calculations which have to be done for every vertex in every frame, it would improve performance in a very significant way.

Modern-day GPU's contain a very versatile pipeline which is optimized for this type of calculations, and has a huge bandwidth to perform them in parallel. Since all of our vertex calculations are independent of the neighboring vertices, accelerating the given algorithms using GPU functionality was a logical next step.

### A. Texture Mapping: a slow 'hack'

Until recently, the only way of using the GPU to implicitly speed up the process of drawing and rendering contours was using a texture map containing a straight line, which was indexed by the radial curvature on the x-axis and the derivative of the radial curvature on the y-axis.

By generating several mipmaps, one could 'fake' the vertex computations and retain a steady line width on all zooming levels. The main problem with this algorithm however, is that the radial curvature and its derivative still have to be computed on the CPU, which hampers the performance significantly. The texture map interpolation also introduces line artifacts.

### B. New algorithm: GPU Shader

In a new algorithm we developed, we use a so-called *Vertex Shader* to compute the curvature information per vertex, on the GPU pipeline. The difficulty was in how to define contour 'lines' in the *Fragment Shader*, which has to handle pixels in parallel.

To test for a *regular* contour line, the regular method is to take the value of  $n \cdot v$  and compare it to a given threshold. This however, results in very thick contour lines where the curvature is very low. By introducing a new limit, we can counter this effect by dividing this value by the curvature:

$$\frac{(n \cdot v)^2}{\kappa_r} < t_c$$

A similar limit can be introduced for suggestive contours, by using the derivative of radial curvature, which is a measure for how fast the curvature changes. Note that all info required for these new limits was already present in the original algorithm: no additional computation is required.

$$\frac{|\kappa_r|}{D_w \kappa_r} < t_{sc}$$

By applying these new limits in which the interpolated radial curvature of the original vertices was accessed in the pixel to decide whether or not it was part of a contour line, we could implement a fast algorithm which uses Object-Space methods to define contour lines on a per-pixel basis.

This resulted in a big performance increase when compared to the traditional CPU methods (even with our previously mentioned improvements), as is demonstrated in the blue bars of the following graph, in which the same benchmark was run for several models of varying complexity.

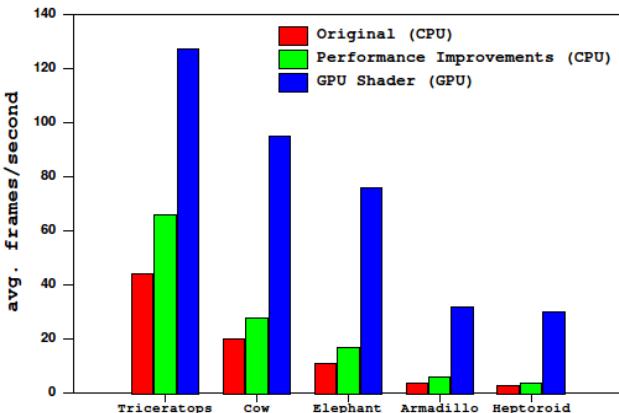


Illustration 5: Performance of GPU shader algorithm compared to traditional CPU algorithms. Results in frames/second

### V. IMAGE SPACE ALGORITHMS

The Image Space-approach to algorithms for contour line extraction and rendering is substantially different from the previously seen Object Space-methods:

First, an intermediate scalar field is built by rendering the original model with a certain type of shading. On this 2D-scalar field, signal analysis is performed to extract the contour lines, which can be overlayed on a render (viewed from the same camera viewpoint) in an extra pass.

This has several advantages: by balancing the work between the intermediate rendering and the signal analysis, the implementation of these algorithms is less complex than for Object Space algorithms. It's also easier to use graphics hardware acceleration, since working with textures and convoluting them is a very common operation.

There's also the notion of *view-dependent-level of detail*: when an object is very small in a frame, the intermediate render will only contain a small percentage of pixels for that object: this results in faster image analysis. For Object Space algorithms, this does not matter: all faces will still be visited.

The disadvantages are an inferior control of line width (since there are no geometrical relations any more in the scalar field) and the absence of a 3d-coordinate list of contour line segments after the algorithm: it works on a per-pixel basis.

#### A. Regular Contour Lines in Image Space

For regular contour lines, two methods were tested:

- *Diffuse rendering with the camera on the light source position*: this results in a scalar field representing  $n \cdot v$  on every pixel. By *thresholding* these values, one can obtain a crude form of contour lines. Line width is a problem.
- *Depth Map rendering with a Sobel Filter [7]*: By applying this edge detection filter on a depth map (very fast to render), one can extract same-width contour lines. There are, however, problems with displaying so called 'inner contours', as demonstrated on the following image. This happens because the depth map values do not change drastically enough, because the rings in the model are close to each other.

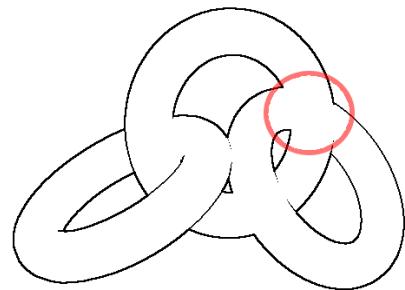


Illustration 6: Sobel filter problems with inner contours

### B. Suggestive Contour Lines in Image Space

Detecting suggestive contour lines on a diffuse-rendered scalar field is performed by using a Radial Filter, proposed in [5]. We want to detect valleys now, which is more complex than detecting differences in color. By using a smoothly-rendered scalar field, however, we can use a simple method to detect these valleys, without having to worry about being noise-robust:

In diffuse-shaded images, valleys are a set of dark pixels cutting through a neighborhood of very bright pixels. By searching in a radius  $r$  around a certain pixel, two conditions should be met for the pixel to be part of a valley (and by consequence, part of a contour line):

- Only a certain percentage of pixels around the pixel should have an intensity smaller than the pixel itself.
- The difference in intensity between the maximum intensity found and the intensity of the current pixel should be smaller than a predefined threshold.

The good news is how this filter also detects *regular* contours, so an extra pass for drawing them is not necessary.

This algorithm was implemented in a vertex shader and has good performance, which is of course influenced by varying the radius of the circle: enlarging the radius makes the algorithm find more valleys, but takes more time.

Model	$r = 2$	$r = 3$	$r = 5$	$r = 10$
Triceratops	81	75	61	45
Cow	65	61	56	40
Elephant	50	45	43	36
Armadillo	22	21	20	18
Heptoroid	29	28	26	22

*Illustration 7: Performance of Image Space Radial Filter algorithm for several radius search values. Results in frames/second.*

We found that varying the output resolution was not the limiting factor in our image-space algorithms: even when resizing to very large resolutions (4096x4096 pixel), the performance degradation was sublinear.

## VI. RESULTS

Some of the results which were generated using the algorithms presented in this article are included in appendix A.

## VII. CONCLUSION

In this article and the related thesis, we have successfully implemented, evaluated and compared existing Object and Image Space algorithms for contour line extraction and rendering.

Performance improvements were developed for existing CPU algorithms, and although these were no avail for rendering more complex models, these algorithms still remain of importance for low-end graphics hardware or embedded systems. Other approaches may be used, for

example: the complex models could be simplified by smoothing them locally.

The suggested GPU-shader Object Space algorithm is a big improvement when it comes to hardware-accelerated contour line drawing: by using radial curvature measures per pixel we retain good control on the line width. By also implementing the existing Image Space algorithms on the GPU, we've provided alternative ways to extract and render contour lines in real-time applications.

## REFERENCES

- [1] D. Decarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella. Suggestive contours for conveying shape. SIGGRAPH, 2003.
- [2] I. Biederman. Recognition-by-components: a theory of human image understanding, Psychological Review vol 94, 1987. 115-147.
- [3] A. Appel. The notion of quantitative invisibility and the machine rendering of solids. Proceedings of ACM National Meeting, 1967. 387-393.
- [4] S. Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. International Symposium on 3D Data Processing Visualization and Transmission, 2004.
- [5] D. Decarlo, A. Finkelstein, and S. Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. NPAR, 2004.
- [6] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. In Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, pages 415–420, August 1997.
- [7] G. Feldman I. Sobel. A 3x3 isotropic gradient operator for image processing. Stanford Artificial Project, 1968.
- [8] J. Koenderink. What does the occluding contour tell us about solid shape? Perception13, 1984. 321-330.
- [9] M.P. DoCarmo. Differential Geometry of Curves and Surfaces. Prentice-Hall, 1976.

## USED TEST MODELS AND BENCHMARK METHOD

The models used to perform the benchmark tests on the algorithms were the following:

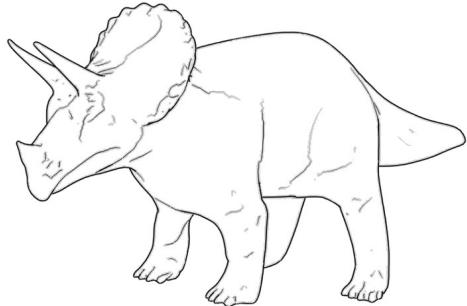
- Triceratops : 80k faces
- Hippo : 120k faces
- Armadillo: 350k faces
- Lucy: 525k faces
- Planck Bust: 530k faces
- Brains: 620k faces

The benchmark method was to rotate the objects in 10.0 seconds over the x, y and z-axis. The average, minimum and maximum framerate for several benchmarks is reported afterwards. These benchmarks were cross-checked with the results of a commercial tool called FRAPS ([www.fraps.com](http://www.fraps.com)).

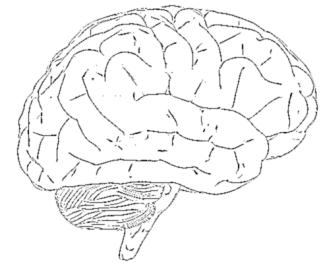
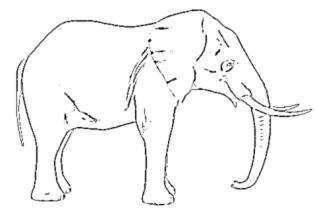
This was performed at a resolution of 512x512 on a Radeon HD4890 GPU, combined with an AMD X4 Phenom II Quad-Core processor, in a virtualized Ubuntu 9.10 OpenGL environment.

## APPENDIX A

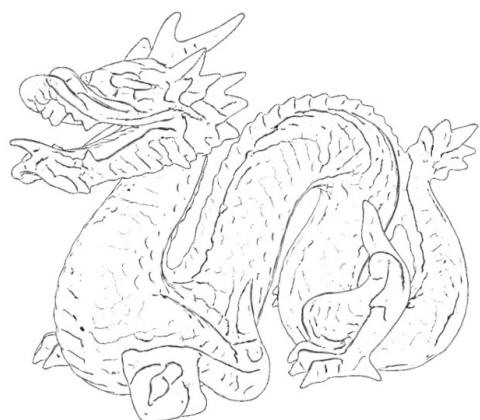
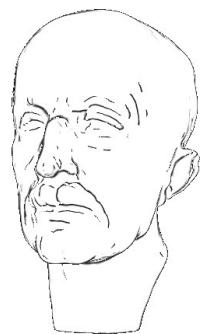
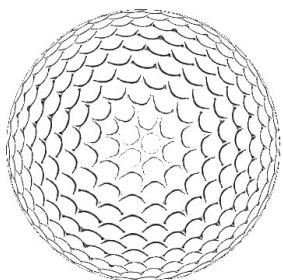
### RESULTS OF CPU-BASED OBJECT SPACE ALGORITHMS



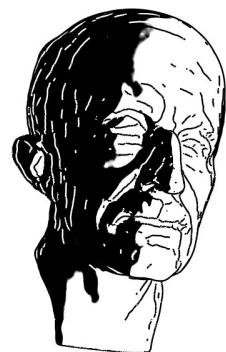
### RESULTS OF IMAGE SPACE ALGORITHMS



### RESULTS OF GPU-BASED OBJECT SPACE ALGORITHMS



### RESULTS OF CPU-BASED SUGGESTIVE HIGHLIGHTS ALGORITHMS



# Bibliografie

- A. Appel. The notion of quantitative invisibility and the machine rendering of solids. *Proceedings of ACM National Meeting*, 1967. 387-393.
- Mike Bailey and Steve Cunningham. *Computer Graphics Shaders: Theory and Practice*. AK Peters, 2009.
- I. Biederman. Recognition-by-components: a theory of human image understanding. *Psychological Review vol 94*, 1987. 115-147.
- R. Cipolla and P.J. Giblin. *Visual motion of curves and surfaces*. Cambridge Univ. Press., 2000.
- D. DeCarlo and S. Rusinkiewicz. Highlight lines for conveying shape. In *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, 2007.
- D. Decarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella. Suggestive contours for conveying shape. *SIGGRAPH*, 2003.
- D. Decarlo, A. Finkelstein, and S. Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. *NPAR*, 2004.
- J. B. Deregowski and S. Dziurawiec. The puissance of typical contours and children's drawings. *Australian Journal of Psychology*, 1996. 98-103.
- M.P. DoCarmo. *Differential geometry of curves and surfaces*. Prentice-Hall, 1976.
- H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, 1971. 623-629.
- J. Gravesen and M. Ungstrup. Constructing invariant fairness measures for surfaces. *Advances in Computational Mathematics 17*, 2002. 67-88.
- H. Guggenheim. *Differential geometry*. Dover, 1977.
- Paul S. Heckbert and Henry P. Moreton. Interpolation for polygon texture mapping and shading. In *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111. Springer-Verlag, 1991.
- D. Hilbert and S. Cohn-Vossen. *Geometry and the imagination*. Springer, 1937.

- Ellen C. Hildreth. Edge detection. 207:187–217, 1985.
- A. Songh Ho. OpenGL vertex buffer object (VBO). [http://www.songho.ca/opengl/gl\\_vbo.html](http://www.songho.ca/opengl/gl_vbo.html), 2005.
- HP. C++ standard template library (STL) specification. <http://www.sgi.com/tech/stl/>, 1994.
- G. Feldman I. Sobel. A 3x3 isotropic gradient operator for image processing. *Stanford Artificial Project*, 1968.
- Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. WYSIWYG NPR: Drawing strokes directly on 3D models. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 21(3):755–762, July 2002.
- Robert D. Kalnins, Philip L. Davidson, Lee Markosian, and Adam Finkelstein. Coherent stylized silhouettes. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 22(3):856–861, July 2003.
- Khronos. GLSL 4.0 specification. <http://www.opengl.org/registry/doc/GLSLangSpec.4.00.8.clean.pdf>, 2009.
- Khronos. OpenGL 4.0 specification. <http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf>, 2006.
- J. Koenderink. What does the occluding contour tell us about solid shape? *Perception* 13, 1984. 321-330.
- Andrew Koenig and Barbara E. Moo. *Accelerated C++: Practical programming by example*. Addison-Wesley Professional, 2000. ISBN 020170353X.
- Yunjin Lee, Lee Markosian, Seungyong Lee, and John F. Hughes. Line drawings via abstracted shading. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 18. ACM, 2007.
- Antonio M. Lopez, Felipe Lumbreras, Joan Serrat, and Juan J. Villanueva. Evaluation of methods for ridge and valley detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21:327–335, 1999. ISSN 0162-8828.
- Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 415–420, August 1997.
- Barbara J. Meier. Painterly rendering for animation. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484, 1996. ISBN 0-89791-746-4.
- F. Miller. *The Art of Sin City*. Dark Horse Comics, 2002. ISBN 1569718180.

- J. Munkres. *Analysis on Manifolds*. Addison-Wesley, 1991.
- M. Van Overveld. An algorithm for polygon subdivision based on vertex normals, 1997.
- D. Paerson and J. Robinson. Visual communications at very low data rates. *IEEE Proceedings*, pages 795–812, April 1985.
- Wang Qiang, Zhigeng Pan, Chen Chun, and Bu Jianjun. Surface rendering for parallel slices of contours from medical imaging. *Computing in Science and Engineering*, 9:32–37, 2007. ISSN 1521-9615.
- R. Raskar. Image processing silhouette edges. *Symposium on Interactive 3D graphics (I3DG)*, 1999.
- D. Rosen. Why you should use OpenGL and not DirectX. URL: <http://blog.wolfire.com/2010/01/Why-you-should-use-OpenGL-and-not-DirectX>, 2010.
- S. Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. *International Symposium on 3D Data Processing Visualization and Transmission*, 2004.
- S. Rusinkiewicz. Line drawings from 3D models. SIGGRAPH Class, 2008.
- S. Rusinkiewicz. Trimesh2 - library for input, output and basic manipulation of 3d meshes. <http://www.cs.princeton.edu/gfx/proj/trimesh2/>, 2006.
- S. Rusinkiewicz, M. Burns, and D. DeCarlo. Exaggerated shading for depicting shape and detail. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 25(3), July 2006.
- D. Scherzer. *Temporal coherence in real-time rendering*. Verlag Dr. Müller, 2010. ISBN 978-3-639-09196-0. URL <http://www.cg.tuwien.ac.at/research/publications/2010/scherzer2010b/>.
- Peter Shirley, Michael Ashikhmin, Michael Gleicher, Stephen Marschner, Erik Reinhard, Kelvin Sung, William Thompson, and Peter Willemsen. *Fundamentals of Computer Graphics, Second Ed.* A. K. Peters, Ltd., Natick, MA, USA, 2005. ISBN 1568812698.
- G. Taubin. Estimating the tensor of curvature of a surface from a polyhedral approximation. *ICCV*, 1995. 902-907.
- Ubisoft. XIII (video game). <http://www.ubi.com/US/Games/Info.aspx?pId=39>, 2003.
- Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983. ISSN 0097-8930.
- Georges A. Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration, 1996.

## Fiche masterproef

*Student:* Jeroen Baert

*Titel:* Contourlijnextractie en weergave in interactieve 3D-toepassingen

*Engelse titel:* Contour line extraction and rendering in interactive 3D applications

*UDC:* 004.5

*Korte inhoud:*

In lijntekeningen maken artiesten/ontwerpers gebruik van verschillende soorten contourlijnen om uiteenlopende eigenschappen van het afgebeelde object weer te geven. We willen deze contourlijnen mathematisch definiëren, berekenen en weergeven met computer graphics-technieken. Het doel is om aan interactieve snelheden complexe modellen te kunnen renderen, waarbij enkel wordt gebruik gemaakt van contourlijnen.

Thesis voorgedragen tot het behalen van de graad van Master in de ingenieurswetenschappen: computerwetenschappen, optie Mens-Machine Communicatie

*Promotor:* Prof. Dr. Ir. Ph. Dutré

*Assessoren:* Prof. Dr. E. Duval  
Dr. Ir. K. Driessens

*Begeleiders:* Dr. B. J. Brown  
Ir. R. Schoukens